Osaamista
ja oivallusta
tulevaisuuden
tekemiseen

Toni Sariola

# 3D-fysiikkamoottori

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

27.5.2020

| | |
|---|---|
| Tekijä<br>Otsikko<br><br>Sivumäärä<br>Aika | Toni Sariola<br>3D-fysiikkamoottori<br><br>47 sivua<br>27.5.2020 |
| Tutkinto | Insinööri (AMK) |
| Tutkinto-ohjelma | Tieto- ja viestintätekniikka |
| Ammatillinen pääaine | Pelisovellukset |
| Ohjaaja | Lehtori Juha Kopu |


Insinöörityön tarkoituksena oli toteuttaa 3D-fysiikkamoottori, jota voidaan käyttää pelisovelluksissa mallintamaan jäykkien kappaleiden liikettä ja niiden välisiä törmäyksiä kolmiulotteisessa tilassa. Fysiikkamoottorille asetettiin vaatimuksiksi olla tarpeeksi tehokas soveltuakseen käytettäväksi yksinkertaisissa pelisovelluksissa sekä helposti muokattavissa vastaamaan erilaisten projektien vaatimuksiin. Työtä varten tutkittiin peleissä tyypillisesti käytettyjen fysiikkamoottorien ohjelmointitoteutuksia ja niiden taustalla toimivaa fysiikkaa.

Työssä ohjelmoitiin 3D-fysiikkamoottori Unity-pelimoottorille käyttäen C#-ohjelmointikieltä. Fysiikkamoottoriin määriteltiin tarvittavat luokat noudattaen Unityn komponenttipohjaista arkkitehtuuria. Toteutuksessa hyödynnettiin joitakin Unityn tarjoamia apufunktioita ja kirjastoja.

Kappaleisiin vaikuttavien voimien ja niistä aiheutuvan liikkeen simuloiminen toteutettiin numeerisesti integroimalla käyttäen semi-implisiittistä Euler-menetelmää. Kappaleiden välisten törmäysten tunnistuksessa käytettävät algoritmit toteutettiin hyödyntämällä työssä luotua geometristen funktioiden kirjastoa. Suorituskyvyn optimoimiseksi törmäysten tunnistus jaettiin karkeaan ja tarkkaan vaiheeseen. Törmäysten käsittelemiseksi päädyttiin impulssipohjaiseen ratkaisumenetelmään. Menetelmän havaittiin kuitenkin aiheuttavan kappaleiden ei toivottua tunkeutumista toisiinsa, jonka korjaamiseksi päädyttiin käyttämään lineaariseksi projisoinniksi kutsuttua tekniikkaa.

Fysiikkamoottorin testausta ja esittelyä varten luotiin esittelysovellus. Sovelluksessa simuloidaan painovoiman alaisten kappaleiden törmäämistä keskenään sekä staattisten, liikkumattomien kappaleiden kanssa.

Insinöörityön lopputuloksena saatiin aikaiseksi toimiva 3D-fysiikkamoottori, joka täyttää sille asetetut vaatimukset. Fysiikkamoottoria voidaan hyödyntää sellaisenaan yksinkertaisissa peliprojekteissa tai laajentaa vaativampia projekteja varten. Fysiikkamoottorin lisäksi syntyi kattava geometriakirjasto, jota voidaan hyödyntää erillään moottorista tai sen laajentamisessa.



| | |
|---|---|
| Avainsanat | fysiikkamoottori, pelifysiikka, simulaatio, törmäystunnistus |

| | |
|---|---|
| Author | Toni Sariola |
| Title | 3D Physics Engine |
| Number of Pages | 47 pages |
| Date | 27 May 2020 |
| Degree | Bachelor of Engineering |
| Degree Programme | Information and Communications Technology |
| Professional Major | Game Applications |
| Instructor | Juha Kopu, Senior Lecturer |

The goal of the project was to create a 3D physics engine that could be used in game applications to model the motion of rigid bodies and collisions between them in a three-dimensional space. Requirements were set for the physics engine to be efficient enough for simple games while being easily modifiable for use in varying types of projects. The technical implementations and the underlying concepts of physics in physics engines were investigated for the project.

In the project a 3D physics engine was created for Unity game engine using the C# programming language. The classes needed for the physics engine were defined obeying the component-based architecture of Unity. Some functions and libraries provided by Unity were used in the implementation.

Simulating the forces acting on the bodies and the resulting motion were implemented with numerical integration using the semi-implicit Euler method. The algorithms for detecting collisions between bodies were implemented utilizing a specifically created library of geometric functions. For performance optimization the collision detection was split into a broad and narrow phase. For handling the collisions, an impulse-based resolution method was used – however, this method was found to cause unwanted penetration between objects. This error was corrected by employing a technique called linear projection.

A demo application was created for assessing and demonstrating the capabilities of the physics engine. The application simulates dynamic bodies falling under the influence of gravity and colliding with each other and static, immovable bodies.

As a result of the project a working 3D physics engine was achieved. The implemented physics engine can be used in simple game applications and further extended for more demanding projects. In addition to the physics engine itself, a comprehensive geometry library was attained, which can be used in expanding the core engine or separate from it.

| | |
|---|---|
| Keywords | physics engine, game physics, simulation, collision detection |

## Sisällys

## Lyhenteet

| | |
|---|---|
| 3D | Three-dimensionality. Kolmiulotteisuus. Tila, jossa paikka määritellään kolmen koordinaatin avulla. |
| AABB | Axis-aligned bounding box. Koordinaattiakseleiden kanssa linjassa oleva suorakulmio tai, kolmessa ulottuvuudessa, suorakulmainen särmiö. |
| SAT | Separating Axis Theorem -algoritmi. Törmäysten tunnistuksessa käytettävä algoritmi. |
| GJK | Gilbert-Johnson-Keerthi-algoritmi. Törmäysten tunnistuksessa käytettävä algoritmi. |

# 1 Johdanto

Insinöörityön tarkoituksena on toteuttaa toimiva 3D-fysiikkamoottori, joka likimääräisesti mallintaa fysikaalisten voimien alaisena liikkuvien jäykkien kappaleiden välisiä törmäyksiä kolmiulotteisessa tilassa. Moottorin on tarkoitus tukea pelien yhteydessä yleisimpiä simulaatioskenaarioita tarpeeksi uskottavasti ja tehokkaasti soveltuakseen käytettäväksi yksinkertaisissa pelisovelluksissa. Toteutuksen pääpainona ei ole niinkään tulosten realistisuus suhteessa todelliseen maailmaan, vaan yleinen käytettävyys pelisovellusten kehityksessä.

Pelien simulaatiotarpeet vaihtelevat usein suuresti lajityypistä, mekaniikoista ja alustasta riippuen, ja tästä syystä usein peliä kehittäessä ei ole lopputuotteen kannalta mielekästä käyttää valmista moottoria. Fysiikkamoottorin ohjelmointi ei kuitenkaan ole yksinkertaista, ja yksittäisen pelin tuotannon resurssit harvoin riittävät tähän. Tästä syystä useimmista kaupallisista ratkaisuista poiketen työssä ei ollut tarkoituksena toteuttaa mahdollisimman geneeristä ja valmista ratkaisua, vaan tarjota käyttäjälle kirjastot ja rajapinnat, joita hyödyntäen voidaan toteuttaa pelikohtaisia fysiikkaratkaisuja ja optimointeja.

Työssä toteutettu fysiikkamoottori ohjelmoitiin C#-kielellä ja tehtiin käytettäväksi yhdessä Unity-pelimoottorin kanssa, joten se noudattaa ohjelmointitoteutuksessaan pitkälti Unityn käytänteitä. Suurin osa työn käsitteistä on kuitenkin pyritty pitämään pelimoottorista tai ohjelmointiympäristöstä riippumattomina. Fysiikkamoottori käyttää toteutuksessaan hyödyksi joitakin pelimoottorista löytyviä apufunktioita, mutta työssä on pyritty selittämään myös näiden toimintaperiaatteet – sekä teorian että käytännön tasolla.

Fysiikka ja törmäyksentunnistus nojaavat vahvasti lineaarialgebraan, joten työn lukijalta odotetaan perustason tuntemusta aiheesta. Erityisesti aiempi osaaminen vektori- ja matriisimatematiikan alalta auttaa käsitteiden ymmärtämisessä.

# 2 Pelimoottorien fysiikkaa

## 2.1 Kinematiikka

Kinematiikka eli geometrinen liikeoppi tutkii kappaleiden liikettä kiinnittämättä huomiota ulkoisten voimien vaikutuksiin. Kinematiikassa keskitytään kappaleen sijaintiin, nopeuteen ja kiihtyvyyteen sekä siihen, kuinka nämä ominaisuudet liittyvät toisiinsa ja muuttuvat ajan mukana. [1; 2, s. 15.]

Kappaleen nopeus $\mathbf{v}$ on vektorisuure, jolla on suuruus ja suunta. Se määritellään tietyllä aikavälillä kappaleen sijainnissa tapahtuneen muutoksen eli siirtymän $\Delta\mathbf{x}$ ja aikavälin pituuden $\Delta t$ osamääränä [1]:

$$\mathbf{v} = \frac{\Delta\mathbf{x}}{\Delta t} \tag{1}$$

Vastaavasti kappaleen kiihtyvyys $\mathbf{a}$ puolestaan määritellään aikavälillä tapahtuneen nopeuden muutoksen $\Delta\mathbf{v}$ ja aikavälin pituuden $\Delta t$ osamääränä [1]:

$$\mathbf{a} = \frac{\Delta\mathbf{v}}{\Delta t} \tag{2}$$

Kaavojen 1 ja 2 määritelmien avulla voidaan edelleen johtaa kaavoissa 3 ja 4 esitetyt kinemaattiset yhtälöt tasaisesti kiihtyvälle etenemisliikkeelle [1]:

$$\mathbf{v} = \mathbf{v}_0 + \mathbf{a}\Delta t \tag{3}$$

$$\mathbf{x} = \mathbf{x}_0 + \mathbf{v}_0\Delta t + \frac{1}{2}\mathbf{a}\Delta t^2 \tag{4}$$

Näiden yhtälöiden avulla on mahdollista laskea kappaleen nopeus $\mathbf{v}$ ja sijainti $\mathbf{x}$ aikavälin lopussa, kun tunnetaan vastaavat arvot $\mathbf{v}_0$ ja $\mathbf{x}_0$ aikavälin alussa. On huomioitava, että nämä yhtälöt pätevät ainoastaan tasaisesti kiihtyvässä liikkeessä, kun kiihtyvyys $\mathbf{a}$ on vakio koko aikavälin ajan. Muuttuva kiihtyvyys ei kuitenkaan muodostu ongelmaksi, kunhan numeerisessa integroinnissa käytettävä aikavälin pituus $\Delta t$ valitaan riittävän pieneksi laskentavirheiden pitämiseksi kohtuullisina. [1.]

Pyörimisliikkeen vastineet nopeudelle ja kiihtyvyydelle ovat kulmanopeus $\boldsymbol{\omega}$ ja kulmakiihtyvyys $\boldsymbol{\alpha}$. Kuten kaavoista 5 ja 6 voidaan havaita, ovat kinemaattiset yhtälöt pyörimisliikkeelle käytännössä identtisiä niiden etenemisliikkeen vastineiden kanssa. [1.]

$$\boldsymbol{\omega} = \boldsymbol{\omega}_0 + \boldsymbol{\alpha}\Delta t \tag{5}$$

$$\boldsymbol{\Omega} = \boldsymbol{\Omega}_0 + \boldsymbol{\omega}_0\Delta t + \frac{1}{2}\boldsymbol{\alpha}\Delta t^2 \tag{6}$$

Yhtälöt 5 ja 6 ilmoittavat kappaleen kulmanopeuden $\boldsymbol{\omega}$ ja kulma-aseman $\boldsymbol{\Omega}$ aikavälin lopussa, kun vastaavat arvot $\boldsymbol{\omega}_0$ ja $\boldsymbol{\Omega}_0$ aikavälin alussa tunnetaan (olettaen, että kulmakiihtyvyys $\boldsymbol{\alpha}$ on vakio koko $\Delta t$:n pituisen aikavälin ajan).

## 2.2 Dynamiikka

Dynamiikka tutkii kappaleisiin vaikuttavien voimien ja niiden aiheuttaman liikkeen välisiä suhteita. Kun kinematiikka auttaa ymmärtämään, millä tavalla kappale liikkuu, dynamiikka kertoo liiketilassa tapahtuvien muutosten syistä. [1; 2, s. 86.]

Myös voima on vektorisuure, jolla on suuruus ja suunta. Mikäli kappaleeseen vaikuttaa ulkoinen voima, se antaa kappaleelle voimavektorin suuntaisen kiihtyvyyden. Kaavasta 7 voidaan havaita, kuinka dynamiikan peruslain eli Newtonin toisen lain mukaisesti voiman $\mathbf{F}$ aiheuttama kiihtyvyys $\mathbf{a}$ on suoraan verrannollinen voimavektorin suuruuteen ja kääntäen verrannollinen kappaleen massaan $m$. [1; 2, s. 31; 3; 4.]

$$\mathbf{a} = \frac{\mathbf{F}}{m} \tag{7}$$

Etenemisliikkeen lisäksi ulkoiset voimat aiheuttavat muutoksia myös kappaleen pyörimistilaan. Voiman kiertävää vaikutusta ilmaisevaa suuretta kutsutaan vääntömomentiksi. Sen suuruus riippuu voiman suuruuden lisäksi myös sen suunnasta sekä voiman vaikutuspisteen ja kappaleen välisestä etäisyydestä. Kaavan 8 mukaisesti vääntömomentti $\boldsymbol{\tau}$ määritellään kappaleen massakeskipisteen voiman vaikutuspisteeseen yhdistävän vektorin $\mathbf{r}$ ja voimavektorin $\mathbf{F}$ ristitulona. [1; 2, s. 36; 3.]

$$\boldsymbol{\tau} = \mathbf{r} \times \mathbf{F} \tag{8}$$

Kun vääntömomentti tunnetaan, voidaan kappaleen kulmakiihtyvyys $\boldsymbol{\alpha}$ laskea kaavan 9 mukaisesti (huomaa vastaavuus etenemisliikkeen kiihtyvyysyhtälön 7 kanssa) [1; 3; 4].

$$\boldsymbol{\alpha} = \mathbf{I}^{-1}\boldsymbol{\tau} \tag{9}$$

Massan sijaan yhtälössä esiintyy kappaleen hitausmomentti $\mathbf{I}$, joka massasta poiketen ei ole skalaari vaan yleisesti $3 \times 3$-matriisina esitetty tensori; merkinnällä $\mathbf{I}^{-1}$ tarkoitetaan tämän matriisin käänteismatriisia. Hitausmomenttitensori kuvaa kappaleen kykyä vastustaa pyörimistä eri akselien ympäri, ja se riippuu kappaleen massan lisäksi sen muodosta. Kaavassa 10 nähdään esimerkkinä laatikkomaisen kappaleen (massa $m$ ja sivut $x$, $y$ ja $z$) paikallisen hitausmomenttitensorin lauseke. [2, s. 56–58; 3; 5, s. 402–403.]

$$\mathbf{I} = \begin{bmatrix} \frac{1}{12}m(y^2+z^2) & 0 & 0 \\ 0 & \frac{1}{12}m(x^2+z^2) & 0 \\ 0 & 0 & \frac{1}{12}m(x^2+y^2) \end{bmatrix} \tag{10}$$

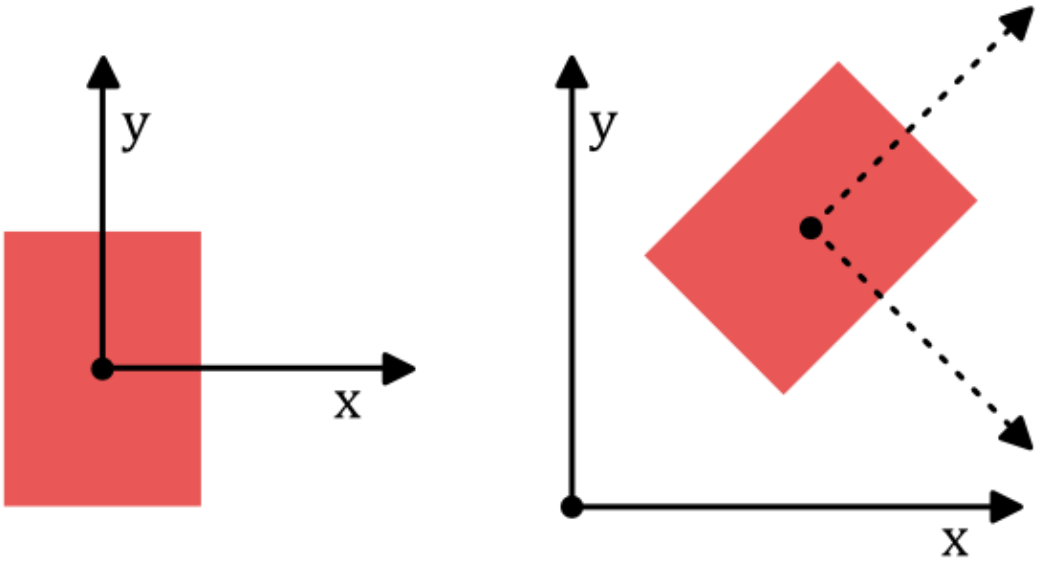
Jäykän kappaleen hitausmomenttitensori on sen paikallisessa koordinaatistossa vakio. Simulaation liikeyhtälöt esitetään kuitenkin maailmakoordinaateissa, joten laskentaa varten tensori tulee muuntaa maailmakoordinaatistoon. Koordinaatistojen välistä muunnosta käsitellään luvussa 2.6.

## 2.3 Jäykkä kappale

Suurin osa peleissä käytetyistä fysiikkaratkaisuista keskittyy kiinteiden, taipumattomien kappaleiden liikkeeseen ja niiden keskinäiseen vuorovaikutukseen – näitä kutsutaan fysiikkamoottorin asiayhteydessä jäykiksi kappaleiksi. Jäykkä kappale voidaan käsittää kokoelmaksi hiukkasia, joiden sijainti kappaleen paikallisessa koordinaatistossa ei muutu. Toisin sanoen jäykkä kappale ei muuta muotoaan liikkuessaan. [1; 3.]

Jäykän kappaleen liikkuessa pyörimättä sen hiukkaset liikkuvat samoilla nopeuksilla, joten kappaleen etenemisliikkeen laskeminen on suhteellisen yksinkertaista. Etenemisliikkeen kannalta kappaletta voidaan laskennallisesti käsitellä sen massakeskipisteessä sijaitsevana yksittäisenä hiukkasena. Sen pyörimisliikkeen laskeminen vaatii kuitenkin hieman erilaista lähestymistapaa. [1; 2, s. 120.]

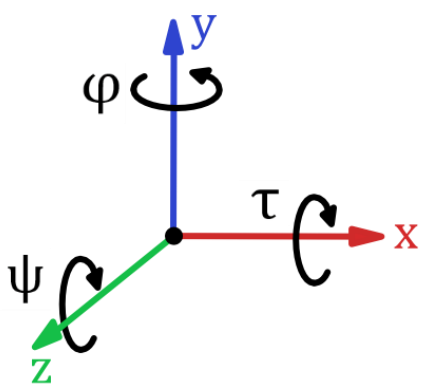
Jäykän kappaleen massakeskipisteen sijaintia voidaan seurata suoraan maailmakoordinaatistossa, mutta pyörimistä varten täytyy ymmärtää käsite kappaleen paikallisesta koordinaatistosta. Kappaleen paikallisen koordinaatiston akselit määritellään suhteessa maailmakoordinaatiston akseleihin, ja sen origo sijaitsee kappaleen massakeskipisteessä (kuva 1). Kappaleen asennon voidaan siis ajatella kuvastavan sen paikallisten koordinaatiston akseleiden eroa verrattuna maailmakoordinaatiston akseleihin. [1.]




Kuva 1. Kappaleen paikallinen koordinaatisto (vasemmalla) suhteessa maailmakoordinaatistoon (oikealla) havainnollistettuna kahdessa ulottuvuudessa [1].

## 2.4 Eulerin kulmat

Kaksiulotteisessa simulaatiossa pyörimisen vapausasteita on ainoastaan yksi, eli pyöriminen tapahtuu ainoastaan yhden akselin ympäri, joten kulma-asema voidaan esittää yksittäisellä skalaariarvolla. Kolmessa ulottuvuudessa pyörimisen vapausasteita on kolme, jolloin kulma-asemaa ja pyörimisliikettä voidaan siis esittää kolmen skalaariarvon avulla. Tähän perustuu menetelmä, jossa kiertymistä kuvataan ns. Eulerin kulmien avulla. Eulerin kulmat $\varphi$, $\tau$ ja $\psi$ esittävät kiertoja 3D-koordinaatiston akseleiden ympäri (kuva 2). Näitä kulmia nimitetään usein myös nyökkäämiseksi, kääntymiseksi ja kallistumiseksi. [1; 5, s. 65–67; 6, s. 54–57.]




τ = nyökkääminen
φ = kääntyminen
ψ = kallistuminen

Kuva 2. Eulerin kulmat, eli kierto koordinaattiakseleiden x, y ja z ympäri, esitettynä nyökkäämisenä, kääntymisenä ja kallistumisena [5, s. 66; 6, s. 56].

Tämä esitystapa on helppo visualisoida ja sisäistää, mutta siihen liittyy ongelma, jonka takia sitä ei voida sellaisenaan käyttää fysiikkasimulaatioissa. Tämä gimbal lock -ongelma ilmenee, kun eräiden yksittäisten kierto-operaatioiden seurauksena kiertoakselit päätyvät yhdensuuntaisiksi, mikä aiheuttaa vapausasteen lukittumisen. Kappaleen asennon seurantaan tulee siis käyttää jotakin muuta tapaa, kuten kvaternioita tai matriiseja. [1; 5, s. 65–67; 6, s. 54–57.]

## 2.5 Kvaterniot

Tässä työssä kappaleen asennon esitystavaksi valittiin kvaternio. Se on abstrakti neliulotteinen matemaattinen objekti, jolla voidaan esittää kiertoa. Mielivaltainen kvaternio $\mathbf{q}$ voidaan mieltää reaaliluvun $w$ ja kolmiulotteisen vektorin $\mathbf{v}$ yhdistelmäksi ja esittää kaavan 11 muodossa. [1; 2, s. 718; 6, s. 58; 7; 8.]

$$\mathbf{q} = [w, \mathbf{v}] = w + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} \tag{11}$$

Tässä $x$, $y$ ja $z$ ovat kvaternion vektoriosan skalaarikomponentit ja $\mathbf{i}$, $\mathbf{j}$ ja $\mathbf{k}$ ovat 3D-koordinaatiston akseleiden suuntaiset yksikkövektorit. Kiertoja esittävät kvaterniot ovat yksikkömittaisia ja toteuttavat siis kaavan 12 normalisaatioehdon. [1; 2, s. 718; 7; 8.]

$$w^2 + x^2 + y^2 + z^2 = 1$$

(12)

Kvaternio, joka esittää kulman $\theta$ suuruista kiertoa yksikkövektorin $\mathbf{n}$ määrittelemän akselin ympäri, voidaan esittää kaavan 13 muodossa [1; 2, s. 721–723; 6, s. 59; 8].

$$\mathbf{q} = [\cos(\theta/2), \sin(\theta/2)\,\mathbf{n}]$$

(13)

Kun kiertokvaternio $\mathbf{q}$ tunnetaan, voidaan alun perin paikkavektorin $\mathbf{p}$ osoittamassa paikassa sijaitsevan pisteen uusi sijainti $\mathbf{p}'$ kierto-operaation jälkeen laskea kaavan 14 yhtälöllä [1; 2, s. 721–723; 6, s. 59–60; 7; 8].

$$\mathbf{p}' = \mathbf{q}\mathbf{p}\mathbf{q}^{-1}$$

(14)

Kaavasta 15 nähdään, kuinka kaavassa 14 tarvittava käänteiskvaternio $\mathbf{q}^{-1}$ voidaan määrittää yksinkertaisesti vaihtamalla sen vektoriosan paikalle tämän vastavektori, eli vaihtamalla komponenttien etumerkit [1; 8].

$$\mathbf{q}^{-1} = [w, -\mathbf{v}] = w - x\mathbf{i} - y\mathbf{j} - z\mathbf{k}$$

(15)

Kun kappaleen kulma-asemaa käsitellään kvaterniona, voidaan kappaleen kierto siis esittää yksinkertaisesti kvaternioiden välisenä tulona. Kiertokvaternioiden välinen tulo on täten kiertojen yhdistelmä. On kuitenkin huomioitava, että kvaterniotulo ei ole vaihdannainen, eli lausekkeen tekijöiden järjestyksellä on merkitystä. Kiertokvaternioilla kerrottaessa kierrot suoritetaan siinä järjestyksessä, kuin ne lausekkeessa esiintyvät. Kvaternioiden $\mathbf{q_1} = [w_1, \mathbf{v_1}]$ ja $\mathbf{q_2} = [w_2, \mathbf{v_2}]$ tulo voidaan esittää kaavassa 16 nähtävässä muodossa. [1; 2, s. 720; 6, s. 58–59; 8.]

$$\mathbf{q_1 q_2} = [w_1, \mathbf{v_1}][w_2, \mathbf{v_2}] = [w_1 w_2 - \mathbf{v_1} \cdot \mathbf{v_2},\ w_1 \mathbf{v_2} + w_2 \mathbf{v_1} + \mathbf{v_1} \times \mathbf{v_2}] \tag{16}$$

Kvaternioita voidaan myös hyödyntää kiertojen lineaariseen interpolointiin. Tästä syystä niitä käytetäänkin laajalti tietokoneanimaatioiden yhteydessä liikkeen pehmentämiseksi ruutujen välillä. Fysiikkamoottorissa kvaternioiden välistä interpolointia hyödynnetään vastaavanlaiseen tarkoitukseen; tähän palataan luvussa 3.3. Kvaternioiden $\mathbf{q_1} = [w_1, \mathbf{v_1}]$ ja $\mathbf{q_2} = [w_2, \mathbf{v_2}]$ välinen lineaarinen interpolointi suoritetaan kaavojen 17 ja 18 mukaisesti. [2, s. 727–729; 6, s. 91–93; 8.]

$$\theta = \arccos(\mathbf{q_1} \cdot \mathbf{q_2}) = \arccos(w_1 w_2 + \mathbf{v_1} \cdot \mathbf{v_2}) \tag{17}$$

$$\mathrm{slerp}(\mathbf{q_1}, \mathbf{q_2}, u) = \frac{\sin[(1-u)\theta]}{\sin(\theta)} \mathbf{q_1} + \frac{\sin(u\theta)}{\sin(\theta)} \mathbf{q_2} \tag{18}$$

Tässä $\theta$ on kvaternioiden välinen kiertokulma, $\mathrm{slerp}$ on kvaternioiden interpolointifunktio ja $u$ on funktiolle parametrina annettava interpolointikohta.

## 2.6 Kiertomatriisit

Kierto koordinaatiston akselin ympäri on lineaarikuvaus, joten se voidaan myös määrittää matriisina. Kiertomatriisi määritellään kolmessa ulottuvuudessa tyypin $3 \times 3$-matriisina $\mathbf{R}$, joka kerrottuna suunta- tai paikkavektorilla $\mathbf{v}$ aiheuttaa kyseisen vektorin kierron halutun akselin ympäri. Kiertomatriiseja käytetään usein esimerkiksi vektorin muuntamiseksi koordinaatistosta toiseen. Matriisilla kierretty vektori $\mathbf{v}'$ määritellään kaavassa 19 esitellyn yhtälön mukaisesti. [9.]

$$\mathbf{v}' = \mathbf{R}\mathbf{v} \tag{19}$$

Kiertomatriisilla voidaan myös kiertää tensoria. Toisen kertaluvun tensorin $\boldsymbol{\sigma}$ kiertäminen tapahtuu kaavan 20 mukaisesti, jossa $\boldsymbol{\sigma}'$ on kierretty tensori ja $\mathbf{R}^{\mathrm{T}}$ merkitsee kiertomatriisin transpoosia. [9.]

$$\boldsymbol{\sigma}' = \mathbf{R}\boldsymbol{\sigma}\mathbf{R}^{\mathrm{T}} \tag{20}$$

Kun tarkastellaan kiertoa erikseen jokaisen 3D-koordinaatiston akselin ympäri, voidaan kierrot esittää matriisimuodossa kaavojen 21, 22 ja 23 mukaisesti [1; 2, s. 713–718; 5, s. 68–69; 8].

$$\mathbf{R}_{\mathrm{x}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta_{\mathrm{x}}) & -\sin(\theta_{\mathrm{x}}) \\ 0 & \sin(\theta_{\mathrm{x}}) & \cos(\theta_{\mathrm{x}}) \end{bmatrix} \tag{21}$$

$$\mathbf{R}_{\mathrm{y}} = \begin{bmatrix} \cos(\theta_{\mathrm{y}}) & 0 & \sin(\theta_{\mathrm{y}}) \\ 0 & 1 & 0 \\ -\sin(\theta_{\mathrm{y}}) & 0 & \cos(\theta_{\mathrm{y}}) \end{bmatrix} \tag{22}$$

$$\mathbf{R}_{\mathrm{z}} = \begin{bmatrix} \cos(\theta_{\mathrm{z}}) & -\sin(\theta_{\mathrm{z}}) & 0 \\ \sin(\theta_{\mathrm{z}}) & \cos(\theta_{\mathrm{z}}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{23}$$

Tässä $\mathbf{R}_{\mathrm{x}}$, $\mathbf{R}_{\mathrm{y}}$ ja $\mathbf{R}_{\mathrm{z}}$ ovat koordinaattiakselien ympäri tapahtuvien kiertojen kiertomatriisit ja $\theta_{\mathrm{x}}$, $\theta_{\mathrm{y}}$ ja $\theta_{\mathrm{z}}$ ovat vastaavat kiertokulmat.

Edellä mainitut matriisit voitaisiin yhdistää yksinkertaisesti niiden keskinäisenä tulona, mutta tämä lähestymistapa tuottaisi vaikeuksia. Jos kierto nimittäin määritellään kolmen kulmamuuttujan $\theta_{\mathrm{x}}$, $\theta_{\mathrm{y}}$ ja $\theta_{\mathrm{z}}$ avulla, päädytään Eulerin kulmien yhteydessä aiemmin mainittuun gimbal lock -ongelmaan. Mielekkäämpi tapa muodostaa kiertoa vastaava matriisi on ilmaista se kvaternioiden tapaan yhtenä kulmana sopivasti valitun akselin ympäri.

Täten myös muunnos esitystapojen välillä – kvaterniosta matriisiksi ja takaisin – yksinkertaistuu huomattavasti. Kiertomatriisi $\mathbf{R}_\mathrm{n}$ mielivaltaisen yksikkövektorina annetun akselin ympäri voidaan ilmaista kaavan 24 esittämällä tavalla. [5, s. 71–76.]

$$\mathbf{R}_\mathrm{n} = \begin{bmatrix} (1-\cos(\theta_\mathrm{n}))x^2 + \cos(\theta_\mathrm{n}) & (1-\cos(\theta_\mathrm{n}))xy + \sin(\theta_\mathrm{n})\,z & (1-\cos(\theta_\mathrm{n}))xz + \sin(\theta_\mathrm{n})\,y \\ (1-\cos(\theta_\mathrm{n}))xy + \sin(\theta_\mathrm{n})\,z & (1-\cos(\theta_\mathrm{n}))y^2 + \cos(\theta_\mathrm{n}) & (1-\cos(\theta_\mathrm{n}))yz + \sin(\theta_\mathrm{n})\,x \\ (1-\cos(\theta_\mathrm{n}))xz + \sin(\theta_\mathrm{n})\,y & (1-\cos(\theta_\mathrm{n}))yz + \sin(\theta_\mathrm{n})\,x & (1-\cos(\theta_\mathrm{n}))z^2 + \cos(\theta_\mathrm{n}) \end{bmatrix} \tag{24}$$

Tässä $\theta_\mathrm{n}$ on kiertokulma ja $x$, $y$ ja $z$ ovat halutun kiertoakselin suuntaisen yksikkövektorin komponentit. Fysiikkamoottori käyttää matriisilla kiertämistä muun muassa hitausmomenttitensorin muuntamiseksi kappaleen paikallisesta koordinaatistosta maailmakoordinaatistoon. Kuten työn aikaisemmassa jäykkiä kappaleita käsittelevässä luvussa 2.3 todetaan, voidaan kappaleen kulma-asema käsittää sen paikallisten akseleiden erona maailmakoordinaatiston akseleihin. Kappaleen paikallisen matriisin muuntaminen maailmakoordinaatistoon voidaan suorittaa sopivasti valitun kiertomatriisin avulla. Kiertooperaatiota vastaava kvaternio, jonka skalaarikomponentit ovat $w, x, y$ ja $z$, voidaan muuntaa kiertomatriisiksi $\mathbf{R}_\mathrm{q}$ kaavassa 25 esitetyllä tavalla [7; 8].

$$\mathbf{R}_\mathrm{q} = \begin{bmatrix} 1-2(y^2+z^2) & 2(xy+wz) & 2(xz-wy) \\ 2(xy-wz) & 1-2(x^2+z^2) & 2(yz+wx) \\ 2(xz+wy) & 2(yz-wx) & 1-2(x^2+y^2) \end{bmatrix} \tag{25}$$

# 3 Fysiikkamoottorin rakenne

## 3.1 Järjestelmän luokkamääritelmä

Fysiikkamoottorin simulaation suorittamisesta ja kappaleiden hallinnoimisesta vastaa fysiikkajärjestelmä (esimerkkikoodi 1). Fysiikkajärjestelmää edustava luokka määritellään Singleton-suunnittelumallin mukaisesti, eli siitä luodusta oliosta voi samanaikaisesti olla ohjelmassa ainoastaan yksi instanssi.

```
public class PhysicsSystem : Singleton<PhysicsSystem>
{
      [SerializeField]
      private PhysicsSettings settings = null;
      public PhysicsSettings Settings { get { return settings; } }

      private List<PhysicsShape> physicsShapes = new List<PhysicsShape>(100);
      private List<ShapePair> collidingPairs = new List<ShapePair>(100);
      private List<CollisionManifold> collisions = new
List<CollisionManifold>(100);

      public void RegisterShape(PhysicsShape shape)
      {
            physicsShapes.Add(shape);
      }

      public void UnregisterShape(PhysicsShape shape)
      {
            physicsShapes.Remove(shape);
      }

      // ...
}
```

Esimerkkikoodi 1. PhysicsSystem-luokka määrittelee fysiikkajärjestelmän, joka vastaa simulaatiosta. Luokka sisältää kentän PhysicsSettings-tietorakenteelle, joka määrittää simulaation parametrit, kuten esimerkiksi sen päivitystaajuuden.

Fysiikkajärjestelmä toimii pääasiallisena käyttörajapintana moottorille ja sisältää muun muassa funktiot kappaleen lisäämiseksi ja poistamiseksi simulaatiosta. Järjestelmä pitää kirjaa simuloitavien kappaleiden lisäksi myös niiden välisistä törmäyksistä ja törmäyksiin liittyvistä kappalepareista.

## 3.2 Kappaleen luokkamääritelmä

Fysiikkamoottori tarvitsee määritelmän tietorakenteelle, joka edustaa jäykkää kappaletta simulaatiossa. Kappale määritellään abstraktina luokkana, josta kaikki erimuotoisia kappaleita edustavat luokat periytetään (esimerkkikoodi 2). Luokalle määritellään kappaleen fysikaalisia ominaisuuksia edustavat kentät, kuten massa, kimmoisuus ja kitka, sekä julkiset ominaisuudet, joilla ulkopuoliset luokat pääsevät tarvittaessa näihin tietoihin käsiksi.

```
public abstract class PhysicsShape : MonoBehaviour
{
      [SerializeField]
      protected float mass = 0f;
      [SerializeField]
      protected float restitution = 0f;
      [SerializeField]
```

```
protected float staticFriction = 0f;
[SerializeField]
protected float dynamicFriction = 0f;

// public properties
// ...
```

Esimerkkikoodi 2. Jäykkää kappaletta edustava PhysicsShape-luokka. Luokka peritään MonoBehaviour-luokasta, joka toimiin Unityssä pohjaluokkana kaikille peliobjektien komponenteille.

Kappaleen fysikaalisten ominaisuuksien lisäksi luokka sisältää kentät sen tilan ja liikkeen seuraamiseksi ja päivittämiseksi (esimerkkikoodi 3). Sijaintia ja asentoa kuvaavien kenttien lisäksi luokka sisältää myös kentät, joita käytetään säilyttämään kappaleen fyysinen tila edelliseltä fysiikkapäivitykseltä. Edellisen fysiikkapäivityksen arvoja käytetään kappaleen visuaalisen tilan interpoloimiseksi ruudunpäivityksen yhteydessä. Interpolointia käsitellään luvussa 3.3.

```
// ...

protected Vector3 position, previousPosition;
protected Vector3 velocity;
protected Vector3 angularVelocity;
protected Vector3 forces;
protected Vector3 torques;
protected Quaternion orientation, previousOrientation;

// ...
```

Esimerkkikoodi 3. Jäykän kappaleen fyysistä tilaa ja liikettä kuvaavat kentät PhysicsShape-luokassa.

Luokka sisältää myös abstraktit hakufunktiot kappaleen muodosta riippuville ominaisuuksille (esimerkkikoodi 4).

```
// ...

public abstract AABB GetBounds();
public abstract Matrix4x4 GetInverseInertiaTensor();

// ...
```

Esimerkkikoodi 4. PhysicsShape-luokalle määritellyt abstraktit hakufunktiot.

GetBounds-funktio laskee ja palauttaa kappaleen rajausmuodon, jota käytetään törmäysten tarkistuksen yhteydessä (esimerkkikoodi 5).

```
//...
```

```
public override AABB GetBounds()
{
      OBB obb = new OBB(position, Extents, Matrix4x4.Rotate(orientation));

      return obb.GetBounds();
}

// ...
```

Esimerkkikoodi 5. GetBounds-funktion toteutus laatikkomaiselle kappaleelle.

GetInverseInertiaTensor-funktio puolestaan palauttaa pyörimisliikkeen laskennassa käytettävän käänteisen hitausmomenttitensorin maailmakoordinaateissa (esimerkkikoodi 6). Hitausmomenttitensori palautetaan käänteisenä, sillä tämä on ainoa muoto, jossa se esiintyy laskennassa. Tensori muunnetaan kappaleen paikallisesta koordinaatistosta maailmakoordinaatistoon kiertämällä se kappaleen asentoa kuvaavasta kvaterniosta muodostetulla matriisilla.

```
public override Matrix4x4 GetInverseInertiaTensor()
{
      Vector3 size = extents * 2f;
      float fraction = (1f / 12f);

      float xSqr = size.x * size.x;
      float ySqr = size.y * size.y;
      float zSqr = size.z * size.z;

      Vector4 i;
      i.x = (ySqr + zSqr) * mass * fraction;
      i.y = (xSqr + zSqr) * mass * fraction;
      i.z = (xSqr + ySqr) * mass * fraction;
      i.w = 1.0f;

      Matrix4x4 tensor = new Matrix4x4();
      tensor[0, 0] = i.x;
      tensor[1, 1] = i.y;
      tensor[2, 2] = i.z;
      tensor[3, 3] = i.w;

      Matrix4x4 rotation = Matrix4x4.Rotate(orientation);
      return rotation * tensor.inverse * rotation.transpose;
}
```

Esimerkkikoodi 6. GetInverseInertiaTensor-funktion toteutus laatikkomaiselle kappaleelle. Orientation-kvaternion muuntamiseksi matriisimuotoon hyödynnetään Unityn 4×4-tyypin matriisia edustavan Matrix4x4-luokan Rotate-funktiota, joka saa parametrikseen kvaterniota edustavan Quaternion-muuttujan ja palauttaa sen matriisimuodossa [10].

Kappaleet voitaisiin lisätä simulaatioon erillisestä koodista, mutta tämä ei ole Unityn komponenttipohjaisen työskentelytavan kannalta optimaalinen ratkaisu. Unityssä MonoBehaviour-komponenteille on määritelty tapahtumafunktiot, joita kutsutaan, kun peliobjekti otetaan käyttöön tai pois käytöstä. Kappaleen lisääminen ja poistaminen simulaatiosta voidaan suorittaa näistä funktioista käsin, jolloin käyttäjän tarvitsee ainoastaan lisätä komponentti peliobjektiin (esimerkkikoodi 7).

```
    // ...

    protected virtual void OnEnable()
    {
        PhysicsSystem.Instance.RegisterShape(this);
    }

    protected virtual void OnDisable()
    {
        PhysicsSystem.Instance.UnregisterShape(this);
    }

// ...
```

Esimerkkikoodi 7. Komponentin OnEnable- ja OnDisable-funktioista kutsutaan PhysicsSystem-luokan kappaleen lisäys- ja poistofunktioita.

Fysiikkakappaleelle määritellystä abstraktista pohjaluokasta voidaan periyttää erimuotoisia kappaleita edustavat aliluokat (esimerkkikoodi 8). Aliluokka toteuttaa pohjaluokassa määritellyt abstraktit funktiot ja sisältää muodon määrittelemiseksi tarvittavat kentät.

```
public class BoxShape : PhysicsShape
{
    [SerializeField]
    protected Vector3 extents = new Vector3();

    public Vector3 Extents { get { return extents; } }

    public override AABB GetBounds()
    {
        OBB obb = new OBB(position, Extents, Matrix4x4.Rotate(orientation));

        return obb.GetBounds();
    }

    public override Matrix4x4 GetInverseInertiaTensor()
    {
        // ...

        Matrix4x4 rotation = Matrix4x4.Rotate(orientation);
        return rotation * tensor.inverse * rotation.transpose;
    }

    protected override void OnDrawGizmos()
```

```
        {
            base.OnDrawGizmos();

            // ...
        }
}
```

Esimerkkikoodi 8. BoxShape-luokka esittää laatikkomaista kappaletta simulaatiossa.

## 3.3 Simulaation päivitys

Simulaatiota askelletaan useimmiten jatkuvasti eteenpäin sovelluksen päivityssilmukassa, ja simulaation tulokset määrittävät ruudulla näkyvien peliobjektien tilan. Joissakin tilanteissa voi olla myös hyödyllistä simuloida aikaa hetkellisesti pidemmälle tulevaisuuteen – esimerkiksi kappaleen liikeradan ennakoimiseksi – tai vaikkapa pysäyttää simulaation päivitys kokonaan, kun fysiikkalaskentaa ei tarvita. Päivitysfunktion avulla voidaan siis simuloida fysiikkakappaleiden tilan muutosta halutun ajan suhteen.

Simulaation päivitys koostuu ensisijaisesti kolmesta eri osasta: numeerisesta integroinnista sekä törmäysten tunnistamisesta ja ratkaisemisesta (esimerkkikoodi 9). Näitä kaikkia käsitellään työssä myöhemmin erikseen luvuissa 4, 5 ja 6.

```
private void StepPhysics(float timeStep)
{
    Integrate(timeStep);
    DetectCollisions();
    ResolveCollisions();
}
```

Esimerkkikoodi 9. StepPhysics-funktio askeltaa simulaatiota eteenpäin parametrina syötetyn aika-arvon verran.

Fysiikkamoottorin aika-askeleelle valitaan arvo, joka määrittää simulaation päivitystaajuuden ja samalla myös sen laskentatarkkuuden. Täytyy siis olla jokin mielekäs tapa valita tämä arvo.

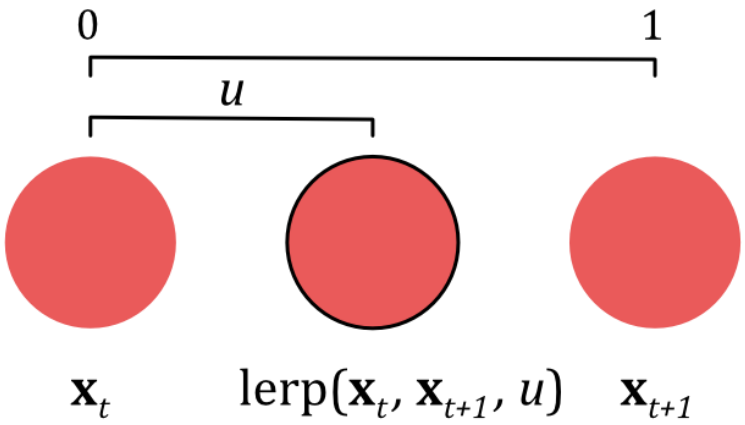
Yksinkertainen ratkaisu olisi käyttää aina edellisestä ruudunpäivityksestä kulunutta aikaa, jotta simulaatio vastaisi mahdollisimman hyvin ruudulla näkyviä tapahtumia. Ruudunpäivitystaajuus kuitenkin vaihtelee ennalta-arvaamattomasti ohjelman suorittamisen aikana, joten tämä saattaisi johtaa tulosten epäjohdonmukaisuuteen suorituskertojen välillä, tai pahimmassa tapauksessa jopa epästabiiliin simulaatioon aika-askeleen ollessa

liian suuri. Jotta simulaation tulokset pysyisivät vakaina ja yhtenäisinä, aika-askeleen halutaankin useimmiten pysyvän vakiona koko ohjelman suorituksen ajan. Tällöin puhutaan kiinteästä aika-askeleesta. Fysiikkamoottorissa simulaation päivitystaajuus määrittää tämän arvon. Tällä tavalla fysiikkasimulaatio eriytetään täysin renderöinnistä, jolloin simulaatio pysyy vakaana ja renderöinti tapahtuu ruudunpäivityksen mukaisesti (esimerkkikoodi 10). [11; 12.]

```
private void Update()
{
      accumulator += Mathf.Min(Time.deltaTime, settings.MaxTimeStep);

      while (accumulator >= settings.TimeStep)
      {
            StepPhysics(settings.TimeStep);
            accumulator -= settings.TimeStep;
      }

      float u = accumulator / settings.TimeStep;
      InterpolateTransforms(u);
}
```

Esimerkkikoodi 10. Päivitysfunktio suoritetaan Unityn Update-funktiossa, jota kutsutaan jokaisella ruudunpäivityksellä. Ruudunpäivitykseen kulunut aika kerätään accumulator-muuttujaan, ja fysiikkapäivitys suoritetaan, kun kerrytetty aika ylittää halutun aika-askeleen. Jäljelle jäävän ajan perusteella lasketaan arvo muuttujalle u, joka syötetään parametrina InterpolateTransforms-funktiolle (esimerkkikoodi 11).

Tämä kuitenkin saattaa aiheuttaa häiritsevää liikkeen epätasaisuutta fysiikkalaskennan päivittyessä epäsynkronoidusti visuaalisen representaation kanssa. Ratkaisu on lineaarinen interpolointi: ruudulla näkyvän peliobjektin tila valitaan kyseisellä hetkellä vallitsevan tilan ja sitä edeltäneen tilan väliltä tavalla, joka riippuu siitä, kuinka kauan aikaa viimeisestä fysiikkapäivityksestä on kulunut (kuva 3). Tällöin kappaleiden liike ruudulla pysyy mahdollisimman pehmeänä. Ainoa haittapuoli tässä on se, että ruudulla tapahtuvat asiat ovat aina hieman jäljessä simulaatiota, mutta kohtalaisella päivitystaajuudella viive jää huomaamattoman pieneksi. [11; 12.]




Kuva 3. Kappaleen sijainnin lineaarinen interpolointi.

Unity-pelimoottorissa peliobjektien sijainnin, kierron ja koon määrittää Transform-komponentti. Esimerkkikoodin 11 funktio interpoloi Transform-komponenttien arvot kappaleiden senhetkisten sekä edeltävällä fysiikkapäivityksellä varastoitujen arvojen välillä. Funktiolle parametrina annettavan muuttujan arvo määrittää välin interpolointikohdan.

```
private void InterpolateTransforms(float u)
{
      for (int i = 0; i < physicsShapes.Count; i++)
      {
            physicsShapes[i].transform.SetPositionAndRotation(
            Vector3.Lerp(physicsShapes[i].PreviousPosition,
physicsShapes[i].Position, u),
            Quaternion.Slerp(physicsShapes[i].PreviousOrientation,
physicsShapes[i].Orientation, u));
      }
}
```

Esimerkkikoodi 11. InterpolateTransforms-funktiossa käytetään hyödyksi pelimoottorista valmiiksi löytyviä, vektoreiden ja kvaternioiden interpolointiin käytettäviä funktioita. Vector3-luokan Lerp-funktio saa parametrikseen kaksi vektorimuuttujaa ja palauttaa näiden välillä lineaarisesti interpoloidun uuden vektorimuuttujan. Slerp-funktio suorittaa vastaavasti kvaternioiden välisen interpoloinnin. Interpoloitavien arvojen lisäksi interpolointifunktioille syötetään parametrina haluttu interpolointikohta; tässä tapauksessa kutsujafunktiolle syötetty parametri u. [13; 14.]

## 4 Numeerinen integrointi

Videopelien fysiikkasimulaatiot toimivat tekemällä kappaleiden tilaan useita pieniä muutoksia fysiikan lakien perusteella. Simulaation yhteydessä voidaan esimerkiksi esittää kappaleen etenemistä tilassa ajan mukana. Jos kappale sijaitsee tietyssä pisteessä ja

kulkee tunnetulla nopeudella tunnettuun suuntaan, voidaan lopullinen sijainti tietyn ajan kuluttua määrittää. Nämä ennusteet suoritetaan matemaattisella tekniikalla, jota kutsutaan numeeriseksi integroinniksi. [15.]

Fysiikkamoottori integroi liikeyhtälöt löytääkseen simuloitaville kappaleille uudet kiihtyvyydet, nopeudet ja siirtymät niihin vaikuttavien voimien ja vääntömomenttien perusteella. Ensimmäiseksi tunnistetaan kaikki kappaleeseen vaikuttavat voimat $\mathbf{F}_i$ sekä vääntömomentit $\boldsymbol{\tau}_i$ ja lasketaan näiden vektorisummat eli nettovoima $\mathbf{F}_{\text{kok}}$ ja nettovääntömomentti $\boldsymbol{\tau}_{\text{kok}}$ (kaavat 26 ja 27). [4.]

$$\mathbf{F}_{\text{kok}} = \sum_{i=1}^{n} \mathbf{F}_i \tag{26}$$

$$\boldsymbol{\tau}_{\text{kok}} = \sum_{i=1}^{n} \boldsymbol{\tau}_i \tag{27}$$

Fysiikkakappaletta edustavalle luokalle määritellään funktiot, joilla siihen voidaan kohdistaa voimia ja vääntömomentteja (esimerkkikoodi 12). Funktiot lisäävät parametrina annetun vektoriarvon integroinnissa käytettävään vektorisummaan eli kasvattavat nettovoimaa tai nettovääntömomenttia.

```
public abstract class PhysicsShape : MonoBehaviour
{
      // ...

      public void AddForce(Vector3 force)
      {
            forces += force;
      }

      public void AddTorque(Vector3 torque)
      {
            torques += torque;
      }

      // ...
}
```

Esimerkkikoodi 12. PhysicsShape-luokan julkiset funktiot, joilla kappaleeseen voidaan kohdistaa voima tai vääntömomentti.

Yhtälöiden 26 ja 27 vektorisummien sekä kappaleen massan $m$ ja hitausmomenttimatriisin käänteismatriisin $\mathbf{I}^{-1}$ avulla voidaan kiihtyvyys $\mathbf{a}$ ja kulmakiihtyvyys $\boldsymbol{\alpha}$ ratkaista kappaleen liikeyhtälöistä (kaavat 28 ja 29) [4].

$$\mathbf{a} = \frac{\mathbf{F}_{\text{kok}}}{m} \tag{28}$$

$$\boldsymbol{\alpha} = \mathbf{I}^{-1}\boldsymbol{\tau}_{\text{kok}} \tag{29}$$

Kun kiihtyvyydet on määritetty, ratkaistaan kappaleen uusi nopeus $\mathbf{v}_{t+1}$ ja kulmanopeus $\boldsymbol{\omega}_{t+1}$ aika-askeleen $\Delta t$ jälkeen sitä edeltäneistä arvoista $\mathbf{v}_t$ ja $\boldsymbol{\omega}_t$ numeerisella integroinnilla (kaavat 30 ja 31) [4].

$$\mathbf{v}_{t+1} = \mathbf{v}_t + \mathbf{a}\Delta t \tag{30}$$

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t + \boldsymbol{\alpha}\Delta t \tag{31}$$

Kappaleen uuden sijainnin $\mathbf{x}_{t+1}$ määrittämiseksi suoritetaan integrointi alkuperäisestä arvosta $\mathbf{x}_t$ edellä laskettua uutta nopeusarvoa $\mathbf{v}_{t+1}$ käyttäen (kaava 32) [4].

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{v}_{t+1}\Delta t \tag{32}$$

Työssä kappaleen asennon esitystavaksi valittiin kvaternio, joten sen päivittäminen eroaa vektorina esitetyn sijainnin päivittämisestä. Ensimmäiseksi tulee laskea asennon muutos $\mathbf{w} = \boldsymbol{\omega}_{t+1}\Delta t$ ja muuntaa se kvaterniomuotoon. Kulmanopeus on vektori, jonka suuruus kertoo kiertokulman ja sen suunta kiertoakselin, joten se sisältää tarvittavat tiedot kiertokvaternion määrittelemiseksi. Asennon muutos voidaan esittää kvaterniona $\mathbf{q}_{\text{w}}$ kaavan 33 mukaisesti.

$$\mathbf{q}_{\mathrm{w}} = \left[\cos\left(\frac{|\mathbf{w}|}{2}\right), \sin\left(\frac{|\mathbf{w}|}{2}\right)\frac{\mathbf{w}}{|\mathbf{w}|}\right] \tag{33}$$

Kappaleen uusi asentokvaternio $\mathbf{q}_{t+1}$ aika-askeleen lopussa saadaan sitä edeltäneen asentokvaternion $\mathbf{q}_t$ ja edellä esitetyn kiertokvaternion välisenä tulona (kaava 34).

$$\mathbf{q}_{t+1} = \mathbf{q}_{\mathrm{w}}\mathbf{q}_t \tag{34}$$

Tässä on huomioitava kvaternioiden laskujärjestys. Kuten työn kvaternioita käsittelevässä vaiheessa luvussa 2.5 todetaan, kvaternioilla kerrottaessa kierrot suoritetaan siinä järjestyksessä, kuin ne esiintyvät lausekkeessa. Koska asennon muutos lasketaan suhteessa maailmakoordinaatteihin, se esiintyy lausekkeessa ensimmäiseksi. Jos kvaternioiden järjestys lausekkeessa käännettäisiin, tapahtuisi pyöriminen kappaleen paikallisten akseleiden ympäri.

Esimerkkikoodissa 13 on esitetty Integrate-funktio, joka parametrina annetun aika-askeleen perusteella integroi liikeyhtälöt simulaation jokaiselle kappaleelle.

```
private void Integrate(float timeStep)
{
      for (int i = 0; i < physicsShapes.Count; i++)
      {
            physicsShapes[i].PreviousPosition = physicsShapes[i].Position;
            physicsShapes[i].PreviousOrientation = physicsShapes[i].Orientation;

            Vector3 acceleration = physicsShapes[i].Forces *
physicsShapes[i].InverseMass;
            physicsShapes[i].Velocity += acceleration * timeStep;
            physicsShapes[i].Position += physicsShapes[i].Velocity * timeStep;

            physicsShapes[i].ClearForces();

            Vector3 angularAcceleration =
physicsShapes[i].GetInverseInertiaTensor().MultiplyVector(physicsShapes[i].Tor
ques);
            physicsShapes[i].AngularVelocity += angularAcceleration * timeStep;

            float angle = physicsShapes[i].AngularVelocity.magnitude * timeStep;
            Vector3 axis = physicsShapes[i].AngularVelocity.normalized;
            float w = Mathf.Cos(angle * 0.5f);
            Vector3 v = Mathf.Sin(angle * 0.5f) * axis;
            physicsShapes[i].Orientation = new Quaternion(v.x, v.y, v.z, w) *
physicsShapes[i].Orientation;
            physicsShapes[i].Orientation.Normalize();
```

```
            physicsShapes[i].ClearTorques();
        }
    }
```

Esimerkkikoodi 13. Integrate-funktio integroi liikeyhtälöt simulaation kappaleille parametrina annetun aika-askeleen perusteella.

Liikeyhtälön ratkaisemiseksi on useita numeerisia integrointitekniikoita. Työn toteutuksessa käytetty tapa on semi-implisiittinen Euler, jota myös suurin osa kaupallisista fysiikkamoottoreista käyttää. Tämän Euler-menetelmän muunnelman ainoa ero alkuperäiseen on, että sijainnin muutos lasketaan päivitettyä nopeusarvoa käyttäen. Semi-implisiittisen Eulerin lisäksi yleisimpiä ovat Verlet- ja Runge-Kutta-menetelmät. [15.]

Euler-menetelmät ovat näistä helpoimpia toteuttaa, mutta ne ovat myös suhteellisen epätarkkoja verrattuna muihin tapoihin. Runge-Kutta-menetelmä on laskennallisesti raskas ja monimutkainen toteuttaa, mutta tarkin näistä kolmesta. Verlet-menetelmällä saavutetaan hyvä tasapaino laskentatehon ja tarkkuuden välillä. Semi-implisiittinen Euler-menetelmä toimii kuitenkin tarpeeksi hyvin useimpiin simulaatiotarpeisiin, kun simulaatiota päivitetään vakiotaajuudella, mikä on vakaan toteutuksen kannalta suositeltavaa. [15; 16.]

# 5 Törmäysten tunnistus

Pelisovellusten asiayhteydessä törmäysten tunnistus ja käsittely mielletään usein fysiikkamoottorin tärkeimmäksi tehtäväksi, ja se on myös työläin implementoida.

Peleissä törmäysten tunnistamisella pyritään säilyttämään illuusio kiinteästä maailmasta. Se estää pelaajahahmoa kävelemästä seinien läpi, sitä voidaan käyttää simuloimaan vastustajan näkölinjaa, ja sen avulla voidaan luoda laukaisualueita pelin tapahtumille. Fysiikkamoottorin asiayhteydessä keskitytään lähinnä siihen, että dynaamiset kappaleet eivät kulje toistensa läpi ja reagoivat törmäyksiin simulaation kannalta uskottavalla tavalla.

Pelisovellusten lisäksi törmäysten tunnistamiselle on tekniikan aloilla useita eri sovellutuksia, kuten tietokoneanimaatiot, robotiikka, virtuaaliset prototyypit ja tekniset simulaatiot [17].

Törmäysten tunnistuksessa pyritään vastaamaan seuraaviin kysymyksiin: Ovatko kaksi kappaletta kosketuksissa? Milloin ne koskettavat? Missä ne koskettavat? Törmäys havaitaan, kun kappaleiden geometriset muodot leikkaavat toisensa. Tällöin tiedetään, että kappaleet ovat kosketuksissa. Usein tämä tieto ei sellaisenaan riitä, vaan tarvitaan enemmän tietoa törmäyksestä, jotta se voidaan käsitellä simulaation vaatimalla tavalla. Näitä tietoja ovat esimerkiksi törmäyksen suunta eli normaali, törmäyksen syvyys ja kontaktipisteet. Törmäykseen liittyvät tiedot kerätään useimmiten tietorakenteeseen, jonka perusteella törmäys käsitellään myöhemmin erillisessä ratkaisuvaiheessa. [17.]

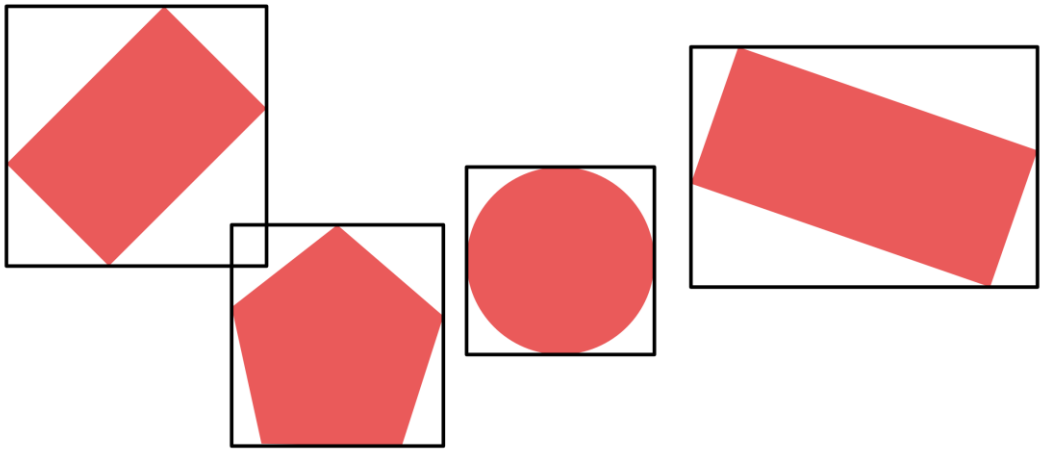
## 5.1 Tunnistusvaiheet

Suorituskykyvaatimukset ovat useimmiten hyvin oleellisessa osassa, kun pelille suunnitellaan järjestelmää törmäysten tunnistusta varten. Etenkin toimintapainotteisissa peleissä, jotta simulaation päivitystaajuus vastaisi mahdollisimman hyvin kuvanpäivitystä, saatetaan simulaation aikana suorittaa törmäyslaskentaa jopa 30–60 kertaa sekunnissa. Kun fysiikkalaskenta muodostaa suuren osan pelin ruudunpäivitykseen kuluvasta ajasta, voi huonosti suunniteltu törmäysten tunnistus koitua keskeiseksi pullonkaulaksi pelin suorituskyvylle. [17.]

Tämän takia törmäysten tunnistaminen jaetaan useimmiten karkeaan ja tarkkaan vaiheeseen. Karkean vaiheen tarkoituksena on määrittää, ovatko kappaleet lähellä toisiaan, ja vasta tarkassa vaiheessa saadaan vastaus siihen, törmäävätkö kappaleet todella. Tarkka törmäystunnistus on useimmiten laskennallisesti raskasta riippuen kappaleiden muotojen monimutkaisuudesta. Tunnistuksen vaiheistamisella vältytään hukkaamasta laskentatehoa turhiin tarkistuksiin monimutkaisten geometristen muotojen välillä. [17; 18.]

Karkeassa tunnistuksessa kappaleet ympäröidään yksinkertaisilla rajausmuodoilla, joiden välistä leikkausta testaamalla saadaan selville, ovatko varsinaiset testattavat kappaleet lähellä toisiaan. Vaikka näiden yksinkertaisten muotojen väliset tarkistukset eivät vielä anna lopullista vastausta, ne ovat huomattavasti kevyempiä laskea kuin esimerkiksi tarkistus kahden mielivaltaisen monitahokkaan välillä. Kolmessa ulottuvuudessa rajausmuotoina käytetään yleisimmin palloja tai AABB:ita. AABB, eli Axis Aligned Bounding

Box, määritellään koordinaattiakseleiden kanssa linjassa olevaksi suorakulmaiseksi särmiöksi. Turhien tarkistusten minimoimiseksi rajausmuoto kappaleelle lasketaan siten, että se on pienin mahdollinen tällainen muoto, joka ympäröi kappaleen kokonaisuudessaan (kuva 4). [17; 18.]



Kuva 4. AABB-muodoilla rajattuja kappaleita [17; 18].

Rajausmuodon valitseminen on kompromissi muodon istuvuuden ja tarkistusalgoritmin laskentanopeuden välillä. Mitä tiiviimmin muoto sopii kappaleen ympärille, sitä vähemmän turhia tarkistuksia joudutaan suorittamaan, ja mitä yksinkertaisempi muoto, sitä vähemmän laskentatehoa yksittäinen tarkistus kuluttaa. Kappaleet simulaatiossa ovat todennäköisemmin erillään toisistaan kuin kontaktissa keskenään, joten tehokasta fysiikkaratkaisua suunnitellessa tulee karkean tarkistuksen optimoimiseen kiinnittää erityisesti huomiota. [17; 18.]

Jos kappaleiden välillä ei tunnisteta mahdollista törmäystä karkeassa vaiheessa, ei niiden välistä törmäystä tarvitse tutkia pidemmälle (esimerkkikoodi 14). Jos karkeassa vaiheessa löydetään potentiaalinen törmäys kappaleiden välillä, jatketaan tarkkaan vaiheeseen, jossa tunnistus suoritetaan kappaleiden varsinaisilla muodoilla.

```
private void DetectCollisions()
{
      collidingPairs.Clear();
      collisions.Clear();
      for (int i = 0; i < physicsShapes.Count; i++)
      {
            for (int j = i + 1; j < physicsShapes.Count; j++)
```

```
            {
                  if (physicsShapes[i].Mass + physicsShapes[j].Mass > 0f)
                  {
                        AABB boundsA = physicsShapes[i].GetBounds();
                        AABB boundsB = physicsShapes[j].GetBounds();

                        if (Geometry.AABBIntersectsAABB(boundsA, boundsB))
                        {
                              ShapePair shapes = new ShapePair(physicsShapes[i],
physicsShapes[j]);
                              CollisionManifold manifold =
GetCollisionManifold(ref shapes);
                              if (manifold.Colliding)
                              {
                                    collidingPairs.Add(shapes);
                                    collisions.Add(manifold);
                              }
                        }
                  }
            }
      }
}
```

Esimerkkikoodi 14. DetectCollisions-funktio testaa sisäkkäisissä silmukoissa kaikki mahdolliset kappaleiden väliset törmäykset. Jos törmäys tunnistetaan, lisätään törmäävä pari ja törmäyksen tiedot listoihin myöhempää käsittelyä varten.

Vaikka työssä toteutetulla karkealla törmäysten tunnistuksella vältytään useilta turhilta tarkistuksilta, ei se ole kuitenkaan ratkaisuna optimaalinen. Karkea tunnistus joudutaan yhä suorittamaan jokaiselle mahdolliselle simulaation kappaleparille. Algoritmin laskennallinen kompleksisuus on siis $O(n^2)$, eli tunnistukseen kuluva aika kasvaa eksponentiaalisesti suhteessa simuloitavien kappaleiden määrään. Kohtuullisen pienillä määrillä kappaleita tämä yksinkertainen implementaatio riittää hyväksyttävän suoritustehon saavuttamiseksi, mutta fysiikkamoottorin joustavuuden ja skaalautuvuuden kannalta algoritmin optimointi olisi suositeltavaa. Karkean vaiheen implementointiin käytetään useimmiten hyödyksi alueellisen jakamisen algoritmeja, joissa tärkeimpänä ajatuksena on potentiaalisesti törmäävien kappaleiden hierarkkinen ryhmittely niiden sijainnin perusteella. Aikarajoitteiden vuoksi optimointeja ei vielä työn kirjoitushetkellä ehditty toteuttamaan, mutta toteutettu törmäysten tunnistus toimii hyvänä pohjana mahdollisille laajennuksille tulevaisuudessa.

## 5.2 Tunnistusalgoritmit

Törmäysten tunnistukseen ei ole olemassa yhtä geneeristä algoritmia, jolla saataisiin laskettua tehokkaasti kaikkien mahdollisten muotojen väliset törmäykset. Riippuen siitä,

mitä muotoja fysiikkamoottorissa halutaan tukea, täytyy kaikille mahdollisille eri muotojen välisille törmäyksille implementoida omat erikoistuneet algoritminsa. Fysiikkamoottoriin toteutettiin törmäysten tunnistusta varten erillinen staattinen kirjasto, johon määriteltiin tietorakenteet tarvittaville geometrisille muodoille sekä funktiot niiden välisten törmäysten testaamiseen. Törmäystä tunnistettaessa valitaan kappaleiden tyyppien perusteella niiden muotojen välistä törmäystä testaava funktio (esimerkkikoodi 15).

```
private CollisionManifold GetCollisionManifold(ref ShapePair shapes)
{
      PhysicsShape shapeA = shapes.A;
      PhysicsShape shapeB = shapes.B;

      CollisionManifold result = new CollisionManifold();
      result.Reset();

      if (shapeA is BoxShape && shapeB is BoxShape)
      {
            result = GetCollisionManifold(shapeA as BoxShape, shapeB as
BoxShape);
      }
      else if (shapeA is SphereShape && shapeB is SphereShape)
      {
            result = GetCollisionManifold(shapeA as SphereShape, shapeB as
SphereShape);
      }
      else if (shapeA is SphereShape && shapeB is BoxShape)
      {
            result = GetCollisionManifold(shapeA as SphereShape, shapeB as
BoxShape);
      }
      else if (shapeB is SphereShape && shapeA is BoxShape)
      {
            result = GetCollisionManifold(shapeB as SphereShape, shapeA as
BoxShape);
            shapes.FlipPair();
      }

      return result;
}
```

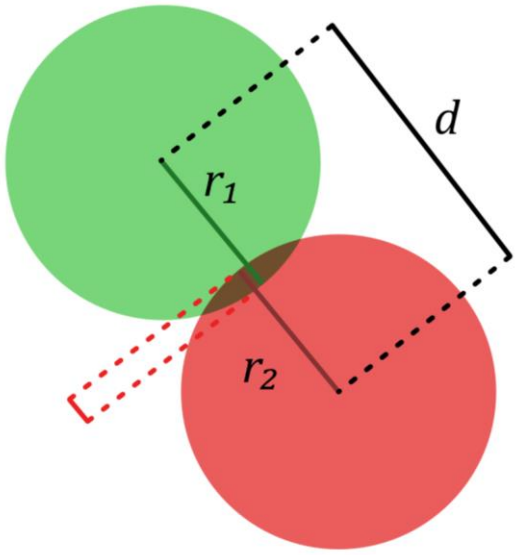
Esimerkkikoodi 15. GetCollisionManifold-funktio palauttaa kappaleparin välisen törmäyksen tiedot sisältävän tietorakenteen. Funktio valitsee kappaleiden muotojen perusteella oikean tunnistusalgoritmin.

Yksinkertaisin tarkistus on kahden ympyrän tai pallon välillä (esimerkkikoodi 16). Jos etäisyys pallojen keskipisteiden välillä on vähemmän kuin niiden säteiden summa, ne leikkaavat toisensa. [5, s. 108; 19.]

```
public static bool SphereIntersectsSphere(Sphere sphere1, Sphere sphere2)
{
      float radii = sphere1.Radius + sphere2.Radius;
      return (sphere1.Center - sphere2.Center).sqrMagnitude <= radii*radii;
}
```

Esimerkkikoodi 16. SphereIntersectsSphere-funktio palauttaa totuusarvon tosi tai epätosi riippuen siitä leikkaavatko sille parametrina syötetyt pallot.

Etäisyyttä laskiessa joudutaan käyttämään raskasta neliöjuurioperaatiota, mutta tämä voidaan välttää vertaamalla etäisyyden neliötä pallojen säteiden summan neliöön. Leikkauksen normaali on yksinkertaisesti pallojen keskipisteiden välisen siirtymän suunta, ja syvyys saadaan vähentämällä etäisyys säteiden summasta (kuva 5). [5, s. 108.]



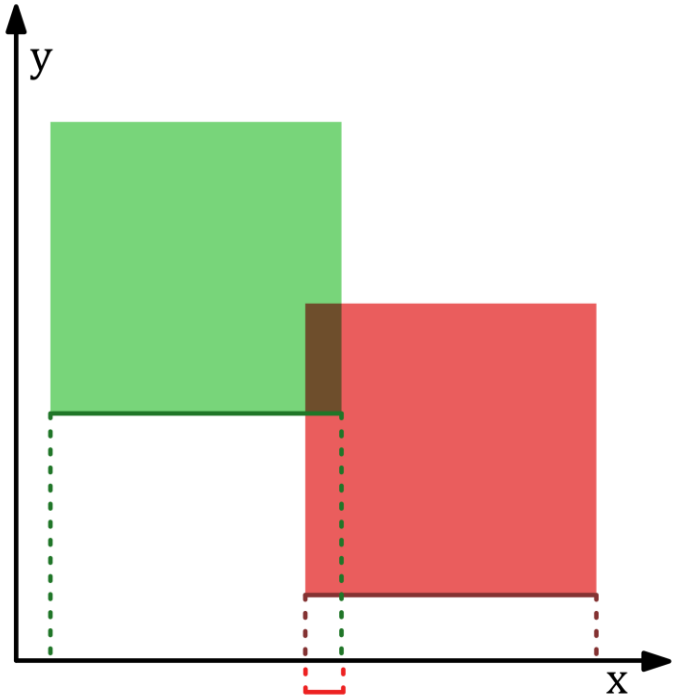

Kuva 5. Kahden ympyrän välinen leikkaus [19].

Toinen yksinkertainen ja yleisimmin karkeissa tunnistuksissa käytetty tarkistus on kahden AABB:n välillä (esimerkkikoodi 17).

```
public static bool AABBIntersectsAABB(AABB aabb1, AABB aabb2)
{
      return (aabb1.min.x <= aabb2.max.x && aabb1.max.x >= aabb2.min.x) &&
             (aabb1.min.y <= aabb2.max.y && aabb1.max.y >= aabb2.min.y) &&
            (aabb1.min.z <= aabb2.max.z && aabb1.max.z >= aabb2.min.z);
}
```

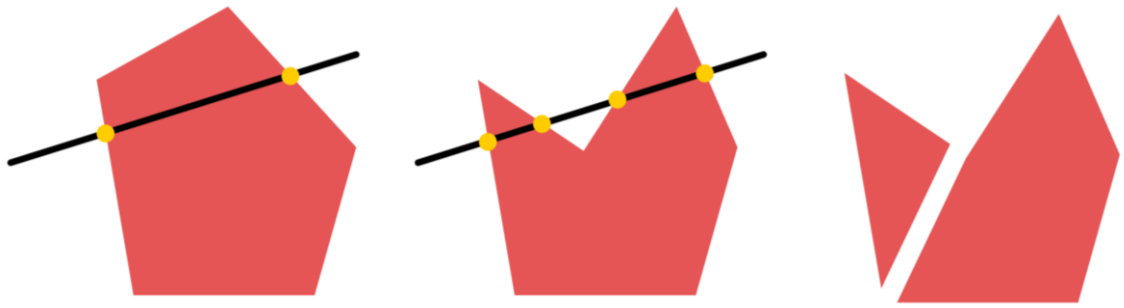
Esimerkkikoodi 17. AABBIntersectsAABB-funktio palauttaa totuusarvona tosi tai epätosi riippuen siitä, leikkaavatko sille parametrina syötetyt AABB:t.

Jotta saadaan selville, leikkaavatko AABB:t, testataan niiden päällekkäisyyttä kaikilla koordinaatiston akseleilla. Jos ne eivät leikkaa jollakin akselilla, ne eivät leikkaa ollenkaan. Törmäyksen syvyys on pienin löydetty päällekkäisyys, ja normaali on akseli, jolta tämä päällekkäisyys löydettiin (kuva 6). [5, s. 183–184; 19.]



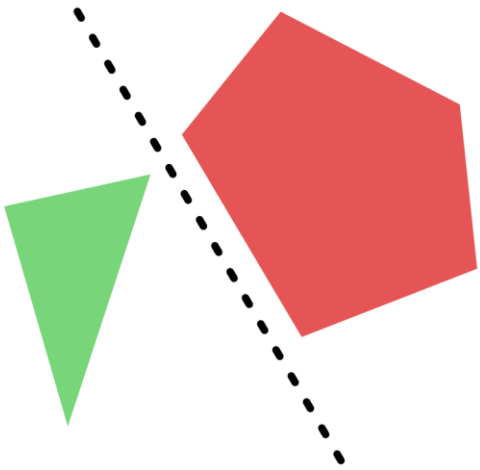

Kuva 6. Kahden suorakulmion välinen leikkaus x-akselilla [5, s. 183].

Kahden AABB:n välinen tarkistus on yksinkertaistettu muoto geneerisestä Separating Axis Theorem- eli SAT-algoritmista, jota hyödynnetään yleensä mielivaltaisten monitahokkaiden välisiin tarkistuksiin. On kuitenkin huomioitava, että SAT toimii sellaisenaan vain, jos testattavat muodot ovat kuperia. Kuten kuvan 7 määritelmästä havaitaan, muoto on kupera, jos mikä tahansa muodon läpi vedettävä viiva leikkaa sen enintään kahdesta eri kohdasta. Jos viiva on mahdollista vetää muodon läpi leikaten sen useammin kuin kahdesti, muoto on kuperan sijasta kovera. Tätä rajoitusehtoa voidaan kuitenkin kiertää helposti hajottamalla kovera muoto useaan kuperaan muotoon. [5, s. 114–117; 18; 20.]



Kuva 7. Kuperan ja koveran muodon määritelmät visualisoituna ja kovera muoto hajotettuna kahteen kuperaan muotoon [20].

SAT:n toimintaperiaatteena on löytää akseli, joka erottaa kaksi muotoa toisistaan. Kuvassa 8 nähdään helppo tapa algoritmin visualisoimiseen kahdessa ulottuvuudessa: jos muotojen välille ei ole mahdollista vetää viivaa, joka erottaa ne toisistaan, eivät muodot voi myöskään leikata toisiaan. Kolmessa ulottuvuudessa viiva edustaa tasoa, jonka normaalivektori on muodot erottava akseli. [18; 20.]



Kuva 8. Kaksi toisistaan viivalla erotettua kuperaa muotoa [20].

Kumpikin muoto projisoidaan jollekin valitulle akselille, jotta saadaan selville, erottaako akseli muodot toisistaan. Muodon projisoiminen akselille suoritetaan etsimällä kaikkien muodon kärkipisteiden sijaintien perusteella minimi- ja maksimikomponentit eli pienimmät ja suurimmat akselin suuntaiset komponentit. Akselin suuntainen komponentti vektorille lasketaan sijainnin paikkavektorin ja akselin välisenä pistetulona (esimerkkikoodi 18). [5, s.116–117; 20.]

```
public void ProjectToAxis(Vector3 axis, out float min, out float max)
{
      Vector3[] vertices = GetVertices();
      float dotP = Vector3.Dot(axis, vertices[0]);
      min = max = dotP;

      for (int i = 1; i < vertices.Length; i++)
      {
            dotP = Vector3.Dot(axis, vertices[i]);
            min = (dotP < min)? dotP : min;
            max = (dotP > max)? dotP : max;
      }
}
```

Esimerkkikoodi 18. ProjectToAxis-funktio projisoi muodon parametrina annetulle akselille ja asettaa heijastuksen minimi- ja maksimikomponentin referensseinä annettuihin min- ja max-muuttujiin.

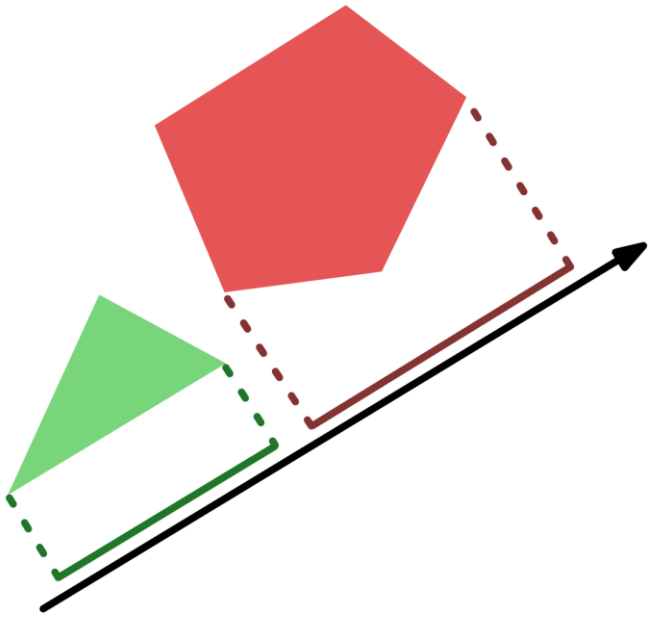
Muotoja projisoidaan eri akseleille, kunnes löydetään projektiot, joilla ei ole päällekkäisyyttä, eli kummankaan minimikomponentti ei ole suurempi kuin toisen maksimikomponentti (esimerkkikoodi 19) [5, s.118–119; 20].

```
public static bool OverlapOnAxis(OBB obb1, OBB obb2, Vector3 axis, out float
overlap)
{
      overlap = 0f;
      float min1, max1, min2, max2;
      obb1.ProjectToAxis(axis, out min1, out max1);
      obb2.ProjectToAxis(axis, out min2, out max2);

      if (min2 > max1 || min1 > max2)
      {
            return false;
      }

      float len1 = max1 - min1;
      float len2 = max2 - min2;
      float min = Mathf.Min(min1, min2);
      float max = Mathf.Max(max1, max2);
      float length = max - min;
      overlap = length - (len1 + len2);

      return true;
}
```

Esimerkkikoodi 19. OverlapOnAxis-funktio, joka projisoi kaksi kiertynyttä suorakulmaista särmiötä parametrina annetulle akselille ja vertaa projektioita keskenään. Jos projektiot ovat päällekkäin, palauttaa funktio totuusarvon tosi ja asettaa lasketun päällekkäisyyden referenssinä annettuun overlap-muuttujaan.

Jos muotojen projektioilla ei ole päällekkäisyyttä yhdelläkin akselilla, eivät muodot leikkaa ja algoritmin suoritus voidaan lopettaa. Kuten kahden AABB:n välisessä tarkistuk-

sessa, löydetään törmäyksen syvyys ja normaali valitsemalla pienin löydetty päällekkäisyys ja sen akseli. Kuvassa 9 esitetään visuaalisesti muotojen projisointia akselille. [18; 19; 20.]



Kuva 9. Kaksi kuperaa muotoa ja niiden projektiot akselille [20].

Mahdollisia erottavia akseleita on ääretön määrä, mutta algoritmin tarvitsee testata vain rajallinen määrä akseleita, jotka määräytyvät testattavien muotojen perusteella. Kolmessa ulottuvuudessa testattavat akselit ovat molempien muotojen tahkojen normaalivektorit sekä niiden kaikkien särmien suuntavektorien keskinäisten ristitulojen muodostamat normaalivektorit. [5, s. 120.]

Muotojen päällekkäisyyden testaaminen keskenään rinnakkaisilla akseleilla antaa täysin saman tuloksen, joten optimointina nämä turhat akselit voidaan jättää testaamatta. Suorakulmaisen särmiön tapauksessa turhien akselien karsimisen jälkeen jää alkuperäisestä 156 akselista testattavaksi ainoastaan 15 (esimerkkikoodi 20). Tällaisissa muotokohtaisissa erikoistapauksissa, joissa testattavien akseleiden määrä on ennalta tiedossa, voidaan erikoistunut algoritmi usein optimoida täysin geneeristä versiota huomattavasti tehokkaammaksi. [5, s.184–185.]

```
public static CollisionManifold GetCollisionManifold(OBB obb1, OBB obb2)
{
     CollisionManifold result = new CollisionManifold();
```

```
        result.Reset();

        Vector3[] axes = new Vector3[15];
        axes[0] = obb1.Orientation.GetColumn(0);
        axes[1] = obb1.Orientation.GetColumn(1);
        axes[2] = obb1.Orientation.GetColumn(2);
        axes[3] = obb2.Orientation.GetColumn(0);
        axes[4] = obb2.Orientation.GetColumn(1);
        axes[5] = obb2.Orientation.GetColumn(2);

        for (int i = 0; i < 3; i++)
        {
            axes[6 + i * 3 + 0] = Vector3.Cross(axes[i], axes[0]);
            axes[6 + i * 3 + 1] = Vector3.Cross(axes[i], axes[1]);
            axes[6 + i * 3 + 2] = Vector3.Cross(axes[i], axes[2]);
        }

        float depth = Mathf.NegativeInfinity;
        Vector3 normal = Vector3.zero;

        for (int i = 0; i < 15; i++)
        {
             float overlap;
             if (!SAT.OverlapOnAxis(obb1, obb2, axes[i], out overlap))
             {
                  result.Reset();
                  return result;
             }
             else if (overlap > depth)
             {
                  depth = overlap;
                  normal = axes[i];
             }
        }

        result.Colliding = true;
        result.Depth = depth;
        result.Normal = normal;

        // ...

        return result;
    }
```

Esimerkkikoodi 20. Funktio, joka testaa törmäyksen kahden kiertyneen suorakulmaisen särmiön välillä SAT-algoritmia käyttäen.

Testattavien akseleiden määrä on suoraan verrannollinen muotojen monimutkaisuuteen, joten algoritmin laskenta voi etenkin kolmiulotteisten muotojen kanssa käydä hyvinkin raskaaksi. Työssä toteutettu moottori käyttää SAT:tä törmäysten tunnistukseen sen yksinkertaisuuden ja toteutuksen suhteellisen helppouden vuoksi, mutta kolmiulotteiseen tarkistukseen on olemassa vaihtoehtoisesti huomattavasti tehokkaampiakin algoritmeja, kuten keksijöittensä mukaan nimetty Gilbert-Johnson-Keerthi- eli GJK-algoritmi. [5, s. 438–440; 21.]

# 6 Törmäysten käsittely

Kun kahden kappaleen välinen törmäys on tunnistettu, tulee seuraavaksi käsitellä tämä törmäys. Törmäys käsitellään kohdistamalla impulssit molempiin kappaleisiin. Impulssilla tarkoitetaan kappaleen liikemäärän, eli sen massan ja nopeuden tulon, välitöntä muutosta. [1; 5, s. 386–392.]

Kappaleiden välisen törmäyksen käsittelyssä käytettävän impulssin suunta ja suuruus selvitetään tunnistusvaiheesta saatujen tietojen perusteella. Törmäyksen ratkaiseminen voidaan suorittaa useita kertoja silmukassa, jolloin simulaation tulosten keskimääräinen tarkkuus kasvaa jokaisella iteraatiolla (esimerkkikoodi 21). Ratkaisemiseen käytettävien iteraatioiden määrän valitseminen on kompromissi simulaation tarkkuuden ja sen suoritustehon välillä.

```
private void ResolveCollisions()
{
      for (int i = 0; i < settings.ImpulseIterations; i++)
      {
            for (int j = 0; j < collisions.Count; j++)
            {
                  for (int k = 0; k < collisions[j].Contacts.Count; k++)
                  {
                        ApplyImpulse(collidingPairs[j], collisions[j], k);
                  }
            }
      }
}
```

Esimerkkikoodi 21. ResolveCollisions-funktio vastaa törmäysten käsittelystä impulssien avulla. Funktio kutsuu ApplyImpulse-funktiota jokaiselle törmäykseen liittyvälle kontaktipisteelle.

Ensimmäiseksi selvitetään kappaleiden välinen suhteellinen nopeus laskemalla niiden nopeuksien erotus. Jos suhteellisen nopeuden suunta on sama kuin törmäysnormaalin suunta, eli pistetulo $\mathbf{v_r} \cdot \mathbf{n} > 0$, etääntyvät kappaleet toisistaan. Tässä tapauksessa kappaleisiin ei kohdisteta impulsseja ja törmäyksen ratkaiseminen voidaan lopettaa. [5, s. 390; 22.]

## 6.1 Kimmoisuus

Kappaleet muuttavat muotoaan törmätessään, ja tähän muodonmuutokseen kuluu liike-energiaa. Törmäysten, joissa liike-energiaa häviää, sanotaan olevan kimmottomia tai plastisia. Törmäystä, jossa liike-energia säilyy ennallaan, kutsutaan puolestaan täysin kimmoiseksi tai elastiseksi. [1.]

Todellisten törmäysten luonne on aina jotain täysin kimmoisan ja täysin kimmottoman välillä. Tätä mallintava parametri tunnetaan sysäyskertoimena, ja sen arvo riippuu törmäävien esineiden materiaalista, rakenteesta ja muodosta. Sysäyskerroin kertoo, kuinka paljon kappaleet menettävät liike-energiaansa törmäyksessä. Simulaatiossa tätä mallinnetaan asettamalla kappalekohtaiset sysäyskertoimet nollan ja yhden väliltä. Törmäyksen sysäyskerroin arvioidaan valitsemalla näistä arvoltaan pienin. On kuitenkin huomioitava, että tämä tapa ei ole fysikaalisesti täysin oikea, sillä todellisuudessa sysäyskertoimen arvo riippuu molempien törmäävien kappaleiden ominaisuuksista. Käytännössä kappalekohtainen sysäyskerroin on kuitenkin kehittäjän kannalta intuitiivinen parametri, ja lopputulos on useimpien pelien simulaatiotarpeisiin nähden tarpeeksi uskottava. [1.]

Törmäyksen yhteydessä kappaleiden suhteellisen nopeuden törmäysnormaalin $\mathbf{n}$ suuntainen komponentti vaihtaa etumerkkiään. Kun törmäyksen sysäyskerroin $e$ otetaan huomioon, voidaan kappaleiden suhteellisen nopeuden törmäystä edeltävän arvon $\mathbf{v}_\mathrm{r}$ ja törmäyksen jälkeisen arvon $\mathbf{v}_\mathrm{r}'$ välille kirjoittaa kaavan 35 mukainen ehto. [5, s. 390–391; 22.]

$$\mathbf{v}_\mathrm{r}' \cdot \mathbf{n} = -e(\mathbf{v}_\mathrm{r} \cdot \mathbf{n}) \tag{35}$$

## 6.2 Impulssien laskenta

Impulssi määritellään liikemäärän muutoksena, joten nopeuden muutos on siis impulssin itseisarvo jaettuna kappaleen massalla. Tämän perusteella kappaleiden nopeudet törmäyksen jälkeen saadaan laskettua kaavojen 36 ja 37 yhtälöiden mukaisesti.

$$\mathbf{v}_{\mathrm{A}}{}' = \mathbf{v}_{\mathrm{A}} + \frac{j\mathbf{n}}{m_{\mathrm{A}}} \tag{36}$$

$$\mathbf{v}_{\mathrm{B}}{}' = \mathbf{v}_{\mathrm{B}} - \frac{j\mathbf{n}}{m_{\mathrm{B}}} \tag{37}$$

Tässä $\mathbf{v}_{\mathrm{A}}$ ja $\mathbf{v}_{\mathrm{B}}$ ovat kappaleiden nopeudet ennen törmäystä ja $\mathbf{v}_{\mathrm{A}}{}'$ ja $\mathbf{v}_{\mathrm{B}}{}'$ törmäyksen jälkeen, $m_{\mathrm{A}}$ ja $m_{\mathrm{B}}$ kappaleiden massat, $j$ impulssin suuruus ja $\mathbf{n}$ törmäyksen normaalivektori. Vaikka molempiin kappaleisiin kohdistuu vastakkaissuuntainen mutta samansuuruinen impulssi, ovat kappaleiden nopeuden muutokset kääntäen verrannollisia niiden massoihin. Kevyempi kappale siis kokee suuremman nopeuden muutoksen kuin raskaampi. [1; 22.]

Käytännön syistä simulaatiossa on hyödyllistä pystyä myös määrittämään täysin liikkumattomia, äärettömän massan omaavia kappaleita. Niitä voivat olla esimerkiksi staattiset rakennukset tai maasto, jota pitkin pelaajahahmo kulkee. Tällaisen kappaleen käänteisen massan arvo on nolla, joten siihen törmäävä kappale vastaanottaa impulssin kokonaisuudessaan. [22.]

Törmäyksen seurauksena kappaleisiin aiheutuu kimpoamisen lisäksi myös pyörimistä. Kappaleille täytyy siis nopeuksien lisäksi ratkaista uudet kulmanopeudet. Lisäksi osoittautuu, että törmäyslaskennan impulssi riippuu täsmällisen törmäyspisteen sijainnista ja törmäävien kappaleiden nopeuksista tässä törmäyskohdassa. Jäykän kappaleen yksittäisen pisteen P nopeus $\mathbf{v}_{\mathrm{P}}$ riippuu kappaleen massakeskipisteen nopeudesta $\mathbf{v}$, kulmanopeudesta $\boldsymbol{\omega}$ ja kappaleen massakeskipisteen pisteeseen P yhdistävästä vektorista $\mathbf{r}$ kaavan 38 mukaisesti. [1; 5, s. 408.]

$$\mathbf{v}_{\mathrm{P}} = \mathbf{v} + \boldsymbol{\omega} \times \mathbf{r} \tag{38}$$

Törmäyksen jälkeiset kulmanopeudet $\boldsymbol{\omega}_{\mathrm{A}}{}'$ ja $\boldsymbol{\omega}_{\mathrm{B}}{}'$ saadaan selville törmäystä edeltäneistä arvoista $\boldsymbol{\omega}_{\mathrm{A}}$ ja $\boldsymbol{\omega}_{\mathrm{B}}$ kaavojen 39 ja 40 mukaisesti [1].

$$\boldsymbol{\omega}_A{}' = \boldsymbol{\omega}_A + \mathbf{I}_A{}^{-1}(\mathbf{r}_A \times j\mathbf{n}) \tag{39}$$

$$\boldsymbol{\omega}_B{}' = \boldsymbol{\omega}_B - \mathbf{I}_B{}^{-1}(\mathbf{r}_B \times j\mathbf{n}) \tag{40}$$

Tässä $\mathbf{I}_A{}^{-1}$ ja $\mathbf{I}_B{}^{-1}$ ovat kappaleiden käänteiset hitausmomenttimatriisit ja $\mathbf{r}_A$ ja $\mathbf{r}_B$ ovat kummankin kappaleen massakeskipisteen törmäyspisteeseen yhdistävät vektorit. Yhtälöissä 36 ja 37 sekä 39 ja 40 tarvittava impulssin suuruus $j$ saadaan kaavasta 41 ($e$ on törmäyksen sysäyskerroin). [1; 5, s. 407–408; 22.]

$$j = \frac{-(1+e)(\mathbf{v}_r \cdot \mathbf{n})}{\frac{1}{m_A} + \frac{1}{m_B} + \mathbf{n} \cdot \left(\mathbf{I}_A{}^{-1}(\mathbf{r}_A \times \mathbf{n}) \times \mathbf{r}_A\right) + \mathbf{n} \cdot \left(\mathbf{I}_B{}^{-1}(\mathbf{r}_B \times \mathbf{n}) \times \mathbf{r}_B\right)} \tag{41}$$

Näillä tiedoilla voidaan ratkaista kappaleiden välinen kitkaton törmäys (esimerkkikoodi 22). Jotta törmäykset simulaatiossa olisivat edes jokseenkin uskottavia, tulee lisäksi simuloida myös kitkan vaikutusta törmääviin kappaleisiin.

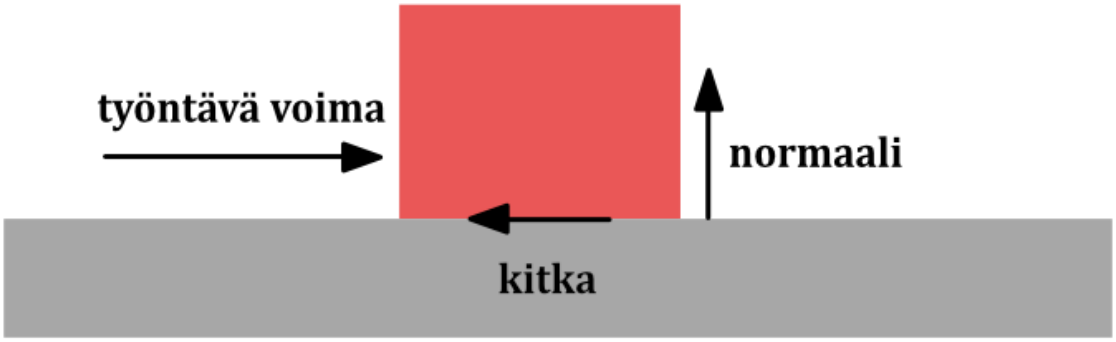
```
private void ApplyImpulse(ShapePair shapes, CollisionManifold manifold, int c)
{
     Vector3 rA = manifold.Contacts[c] - shapes.A.Position;
     Vector3 rB = manifold.Contacts[c] - shapes.B.Position;

     Matrix4x4 iA = shapes.A.GetInverseInertiaTensor();
     Matrix4x4 iB = shapes.B.GetInverseInertiaTensor();

     Vector3 relativeVelocity = (shapes.A.Velocity +
Vector3.Cross(shapes.A.AngularVelocity, rA)) - (shapes.B.Velocity +
Vector3.Cross(shapes.B.AngularVelocity, rB));

     float velocityAlongNormal = Vector3.Dot(relativeVelocity,
manifold.Normal);
     if (velocityAlongNormal > 0f) return;

     float restitution = Mathf.Min(shapes.A.Restitution,
shapes.B.Restitution);
     float inverseMassSum = shapes.A.InverseMass + shapes.B.InverseMass;
     float j = -(1 + restitution) * velocityAlongNormal - correction;

     Vector3 dA = Vector3.Cross(iA.MultiplyVector(Vector3.Cross(rA,
manifold.Normal)), rA);
     Vector3 dB = Vector3.Cross(iB.MultiplyVector(Vector3.Cross(rB,
manifold.Normal)), rB);
     float denominator = inverseMassSum + Vector3.Dot(manifold.Normal, dA +
dB);
     j /= (denominator == 0f) ? 1f : denominator;
```

```
        if (manifold.Contacts.Count > 0)
        {
            j /= manifold.Contacts.Count;
        }

        Vector3 normalImpulse = j * manifold.Normal;

        shapes.A.Velocity += shapes.A.InverseMass * normalImpulse;
        shapes.B.Velocity -= shapes.B.InverseMass * normalImpulse;

        shapes.A.AngularVelocity += iA.MultiplyVector(Vector3.Cross(rA,
normalImpulse));
        shapes.B.AngularVelocity -= iB.MultiplyVector(Vector3.Cross(rB,
normalImpulse));

        // ...
}
```

Esimerkkikoodi 22. ApplyImpulse-funktion osuus törmäysimpulssin laskennalle. Laskenta suoritetaan erikseen jokaiselle törmäykseen liittyvälle kontaktipisteelle, joten impulssin suuruus jaetaan kontaktipisteiden lukumäärällä.

## 6.3 Kitka

Kitka eli liikevastus kohdistetaan simulaatiossa kappaleisiin erillisenä impulssina. Törmäysnormaalin suuntaisella impulssilla pyritään estämään kappaleita läpäisemästä toisiaan, kun taas kitkaimpulssi vaikuttaa kosketuspintojen välisen liikkumisen estämiseksi. Kitka on siis törmäyspinnan suuntaista liikettä vastustava voima. Kuvassa 10 visualisoidaan kitkan vaikutusta kappaleen ja tason väliseen liikkeeseen. [1; 23.]




Kuva 10. Kitkavoiman vaikutuksen alaisena tasoa pitkin liikkuva kappale [1].

Kun kappaleiden suhteellisesta nopeusvektorista vähennetään normaalin suuntainen komponentti, jäljelle jää ainoastaan pinnan suuntainen komponentti. Tämän vektorin suunta on törmäyksen tangentti eli vektori, joka osoittaa suhteellisen nopeuden pinnan myötäiseen suuntaan. Kitkaimpulssin tulee siis osoittaa tangentin vastavektorin suuntaan vastustaakseen pinnan suuntaista liikettä. Lasketun vektorin itseisarvo kertoo kitkaimpulssin maksimisuuruuden. [1; 23.]

Kuten kaavasta 42 voidaan havaita, on yhtälö kitkaimpulssille $j_\mathrm{t}$ suurimmaksi osaksi sama kuin törmäysimpulssille. Erona on, että törmäysnormaalin tilalla on törmäystangentti $\mathbf{t}$. [1; 5, s. 408; 23.]

$$j_\mathrm{t} = \frac{-(1+e)(\mathbf{v}_\mathrm{r} \cdot \mathbf{t})}{\frac{1}{m_\mathrm{A}} + \frac{1}{m_\mathrm{B}} + \mathbf{t} \cdot \left(\mathbf{I}_\mathrm{A}^{-1}(\mathbf{r}_\mathrm{A} \times \mathbf{t}) \times \mathbf{r}_\mathrm{A}\right) + \mathbf{t} \cdot \left(\mathbf{I}_\mathrm{B}^{-1}(\mathbf{r}_\mathrm{B} \times \mathbf{t}) \times \mathbf{r}_\mathrm{B}\right)} \tag{42}$$

Kun levossa olevat kappaleet ovat kosketuksissa toisiinsa, kuluu niiden liikkeelle saamiseen todellisuudessa tietty määrä energiaa. Kun kappaleen saa pintaa pitkin liikkeelle, se on usein helpompaa myös pitää liikkeessä. Tätä liikkeelle lähtöä vastustavaa kitkaa kutsutaan lepokitkaksi. Myös lepokitkaa voidaan kuvata yksinkertaisella matemaattisella mallilla. [1; 23.]

Jos ratkaistun kitkaimpulssin suuruus on pienempi kuin lepokitkakertoimen ja törmäysimpulssin tulo, voidaan olettaa kappaleen olevan levossa. Tällöin impulssi käytetään kokonaisuudessaan. Jos taas kitkaimpulssin suuruus ylittää tämän kynnysarvon, kerrotaan se liikekitkakertoimella. Liikekitka- ja lepokitkakertoimet voidaan laskea kappalekohtaisten kitkakerrointen perusteella kaavassa 43 esitetyllä tavalla.

$$\mu = \sqrt{\mu_\mathrm{A} + \mu_\mathrm{B}} \tag{43}$$

Tässä $\mu$ on törmäyksen kitkakerroin, kun $\mu_\mathrm{A}$ ja $\mu_\mathrm{B}$ ovat kappalekohtaiset kitkakertoimet. On huomioitava, että yksittäisen kappaleen kitkakerroin on kappalekohtaisen sysäyskertoimen tavoin fysikaalisesti kyseenalainen mutta käytännöllinen parametri fysikaalisen ilmiön likimääräiseksi mallintamiseksi. Vaihtoehtoinen, paremmin todellisuutta kuvaava

ratkaisu kitkakertoimen valitsemiseksi voisi olla esimerkiksi eri materiaalipareista koostuva taulukko. [1; 5, s. 391–392; 23.]

Kaavoista 44–47 nähdään, kuinka kitkaimpulssit kohdistetaan kappaleisiin käytännössä täysin samalla tavalla kuin törmäysimpulssit ratkaisun aikaisemmassa vaiheessa [1; 5, s. 391–408; 23].

$$\mathbf{v}_{\mathrm{A}}{}' = \mathbf{v}_{\mathrm{A}} + \frac{j_{\mathrm{t}}\mathbf{t}}{m_{\mathrm{A}}} \tag{44}$$

$$\mathbf{v}_{\mathrm{B}}{}' = \mathbf{v}_{\mathrm{B}} - \frac{j_{\mathrm{t}}\mathbf{t}}{m_{\mathrm{B}}} \tag{45}$$

$$\boldsymbol{\omega}_{\mathrm{A}}{}' = \boldsymbol{\omega}_{\mathrm{A}} + \mathbf{I}_{\mathrm{A}}{}^{-1}(\mathbf{r}_{\mathrm{A}} \times j_{\mathrm{t}}\mathbf{t}) \tag{46}$$

$$\boldsymbol{\omega}_{\mathrm{B}}{}' = \boldsymbol{\omega}_{\mathrm{B}} - \mathbf{I}_{\mathrm{B}}{}^{-1}(\mathbf{r}_{\mathrm{B}} \times j_{\mathrm{t}}\mathbf{t}) \tag{47}$$

Näillä tiedoilla voidaan ratkaista kappaleiden välinen törmäys kitkan vaikuttaessa simulaatiossa (esimerkkikoodi 23).

```
private void ApplyImpulse(ShapePair shapes, CollisionManifold manifold, int c)
{
      // ...
      Vector3 tangent = relativeVelocity - velocityAlongNormal *
manifold.Normal;
      tangent.Normalize();

      float velocityAlongTangent = Vector3.Dot(relativeVelocity, tangent);
      float jt = -(1 + restitution) * velocityAlongTangent;

      dA = Vector3.Cross(iA.MultiplyVector(Vector3.Cross(rA, tangent)), rA);
      dB = Vector3.Cross(iB.MultiplyVector(Vector3.Cross(rB, tangent)), rB);
      denominator = inverseMassSum + Vector3.Dot(tangent, dA + dB);
      jt /= (denominator == 0f) ? 1f : denominator;

      if (manifold.Contacts.Count > 0)
      {
            jt /= manifold.Contacts.Count;
      }
```

```
      Vector3 frictionImpulse;
      float staticFriction = Mathf.Min(Mathf.Sqrt(shapes.A.StaticFriction *
shapes.B.StaticFriction), 1f);
      if (Mathf.Abs(jt) < j * staticFriction)
      {
            frictionImpulse = jt * tangent;
      }
      else
      {
            float dynamicFriction = Mathf.Sqrt(shapes.A.DynamicFriction *
shapes.B. DynamicFriction);
            frictionImpulse = jt * dynamicFriction * tangent;
      }

      shapes.A.Velocity += shapes.A.InverseMass * frictionImpulse;
      shapes.B.Velocity -= shapes.B.InverseMass * frictionImpulse;

      shapes.A.AngularVelocity += iA.MultiplyVector(Vector3.Cross(rA,
frictionImpulse));
      shapes.B.AngularVelocity -= iB.MultiplyVector(Vector3.Cross(rB,
frictionImpulse));
}
```

Esimerkkikoodi 23. ApplyImpulse-funktion osuus kitkaimpulssin laskennalle. Huomattavin ero törmäysimpulssin laskentaan on ehtolause kitkakertoimen valitsemiseksi.

## 6.4 Sijainnin korjaus

Impulssipohjainen ratkaisuvaihe ei käsittele kappaleiden sijainteja suoraan, vaan muokkaa ainoastaan niiden nopeuksia ja kulmanopeuksia törmäysten ratkaisemiseksi. Jos ratkaisua käytettäisiin sellaisenaan, tämä koituisi simulaatiossa ongelmaksi painovoiman kaltaisten jatkuvien voimien tapauksessa.

Tässä työssä käsitelty törmäysten tunnistus tapahtuu vasta, kun kappaleet ovat jo liikkuneet, eli kun törmäys tunnistetaan, tiedetään, että kappaleet ovat jo jonkin verran toistensa sisällä. Jos kappaleita yhteen työntävien voimien aiheuttama liikemäärän muutos on suurempi kuin törmäyslaskennan antama impulssi, jatkavat kappaleet liikettä toisiaan kohti. Tämä ilmenee etenkin dynaamisen ja staattisen kappaleen välisessä törmäyksessä. Painovoiman alaisen kappaleen ollessa simulaatiossa jatkuvassa liikkeessä vasten staattista lattiaa voidaan ilman korjaavia toimenpiteitä havaita kappaleen vajoamista lattiaan. Ongelman epäfysikaalisen luonteen takia ei sen ratkaisemiseksi ole myöskään täysin fysikaalisesti tyydyttävää ratkaisua. Käytännöllinen ratkaisu on kuitenkin yksinkertaisesti toteutettavissa.

Törmäyksen tunnistuksesta saatujen tietojen perusteella kappaleita voidaan siirtää tietty prosentuaalinen määrä törmäyksen syvyydestä ulos törmäyksestä. Tätä tekniikkaa kutsutaan lineaariseksi projisoinniksi. Projisointi voidaan suorittaa täysin erillään törmäyksen ratkaisuvaiheesta muokkaamalla kappaleiden sijainteja suoraan, tai vaihtoehtoisesti ottamalla korjaus huomioon impulssilaskennassa, jolloin kappaleisiin kohdistuvia impulsseja painotetaan normaalin suuntaisesti suhteessa törmäyksen syvyyteen. Työn toteutuksessa päädyttiin ratkaisuista ensimmäiseen, sillä käytännön kokeiluissa sillä saavutettiin vakaampi lopputulos. [5, s. 393–398; 22.]

Sijainninkorjausfunktiota (esimerkkikoodi 24) kutsutaan, kun kaikki fysiikkapäivityksen aikana tapahtuneet törmäykset on ratkaistu nopeuksien suhteen.

```
private void CorrectPositions()
{
      for (int i = 0; i < collisions.Count; i++)
      {
            LinearProjection(collidingPairs[i], collisions[i]);
      }
}
```

Esimerkkikoodi 24. CorrectPositions-funktio käy silmukassa läpi kaikki törmäykset ja kutsuu LinearProjection-funktiota, joka saa parametrikseen kappaleparin sekä törmäykseen liittyvät tiedot.

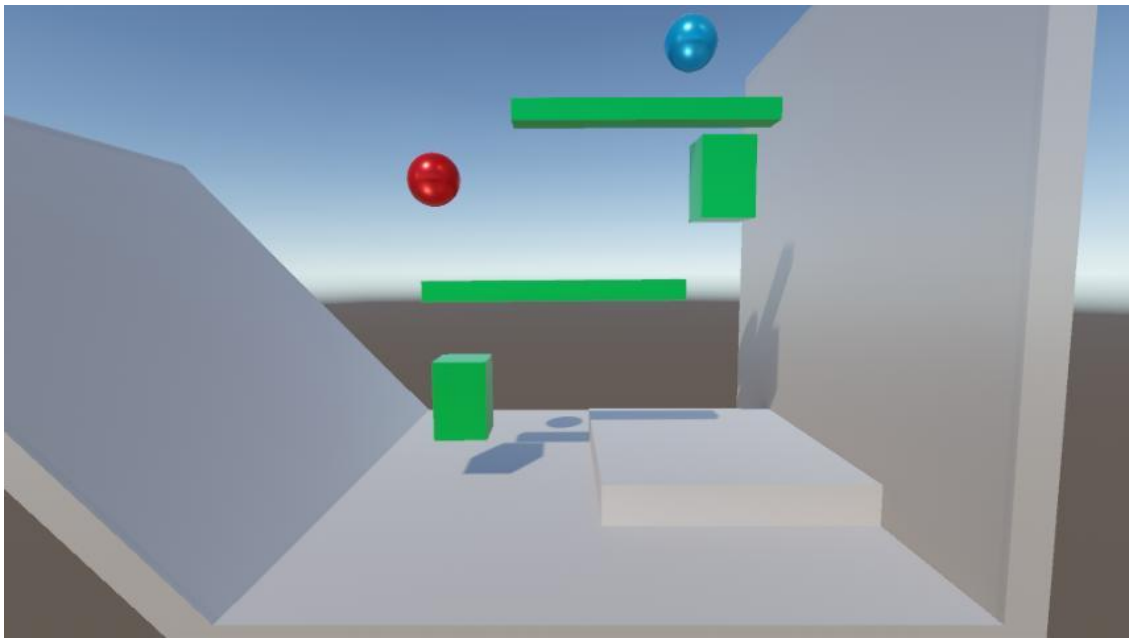
Kappaleille suoritetaan lineaarinen projisointi törmäyksen tietoja käyttäen (esimerkkikoodi 25). Kappaleiden välistä massasuhdetta noudattaen kappaleita siirretään kääntäen suhteessa niiden massoihin. Jotta projisointi aiheuttaisi mahdollisimman vähän näkyvää tärinää kappaleiden poukkoillessa sisään törmäykseen ja ulos törmäyksestä, asetetaan simulaatiolle pieni kynnysarvo, joka määrittelee, kuinka paljon kappaleiden annetaan liikkua toistensa sisään, ennen kuin niitä siirretään. [5, s. 393–398; 22.]

```
private void LinearProjection(ShapePair shapes, CollisionManifold manifold)
{
      float depth = Mathf.Min(manifold.Depth + settings.PenetrationTolerance,
0f) * settings.LinearProjectionPercent;
      float inverseMassSum = shapes.A.InverseMass + shapes.B.InverseMass;
      float scalar = depth / inverseMassSum;

      Vector3 projection = scalar * manifold.Normal;

      shapes.A.Position -= shapes.A.InverseMass * projection;
      shapes.B.Position += shapes.B.InverseMass * projection;
}
```

Esimerkkikoodi 25. LinearProjection-funktio suorittaa kappaleparille lineaarisen projisoinnin törmäyksen tietoja käyttäen.

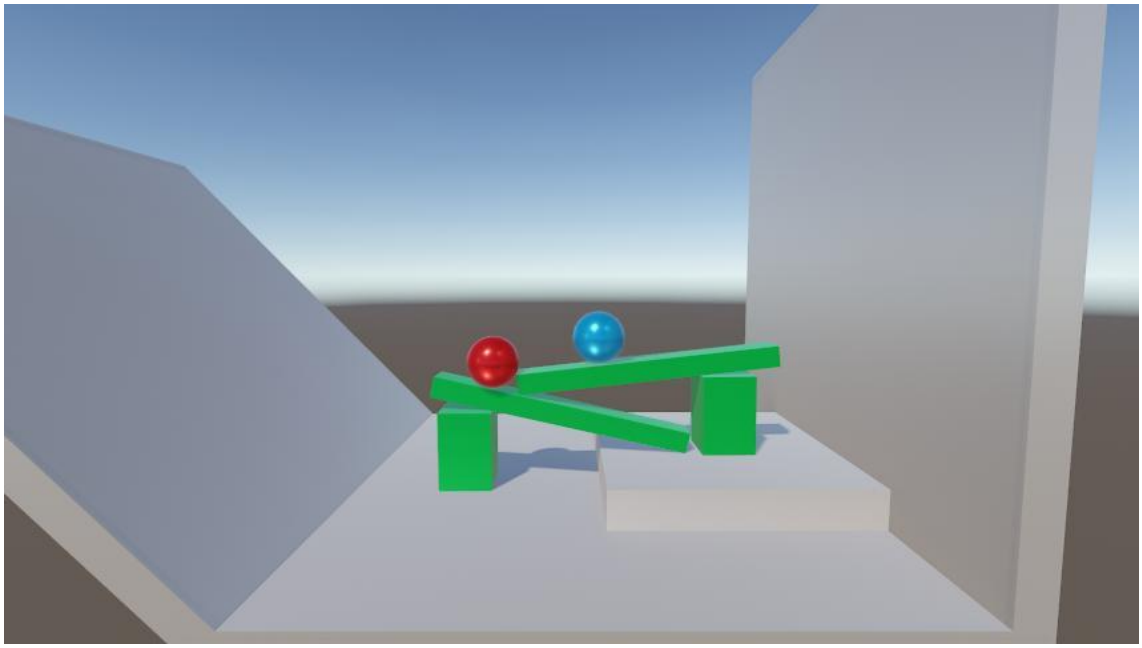
# 7 Työn tulokset

Insinöörityön tulosten esittämistä varten luotiin esittelysovellus, joka esittelee fysiikkamoottorin kykyä simuloida erimuotoisten jäykkien kappaleiden käyttäytymistä voimien alaisena sekä kappaleiden keskinäisiä vuorovaikutuksia niiden törmäillessä toisiinsa. Sovellukseen määriteltiin näkymä, joka käsittää muutamasta staattisesta äärettömän massan omaavasta kappaleesta muodostetun avonaisen alustan sekä painovoiman alaisena putoavia dynaamisia kappaleita. Kuvassa 11 nähdään esittelysovelluksen alkuasetelma, jossa dynaamiset kappaleet eivät ole vielä lähteneet liikkeelle.



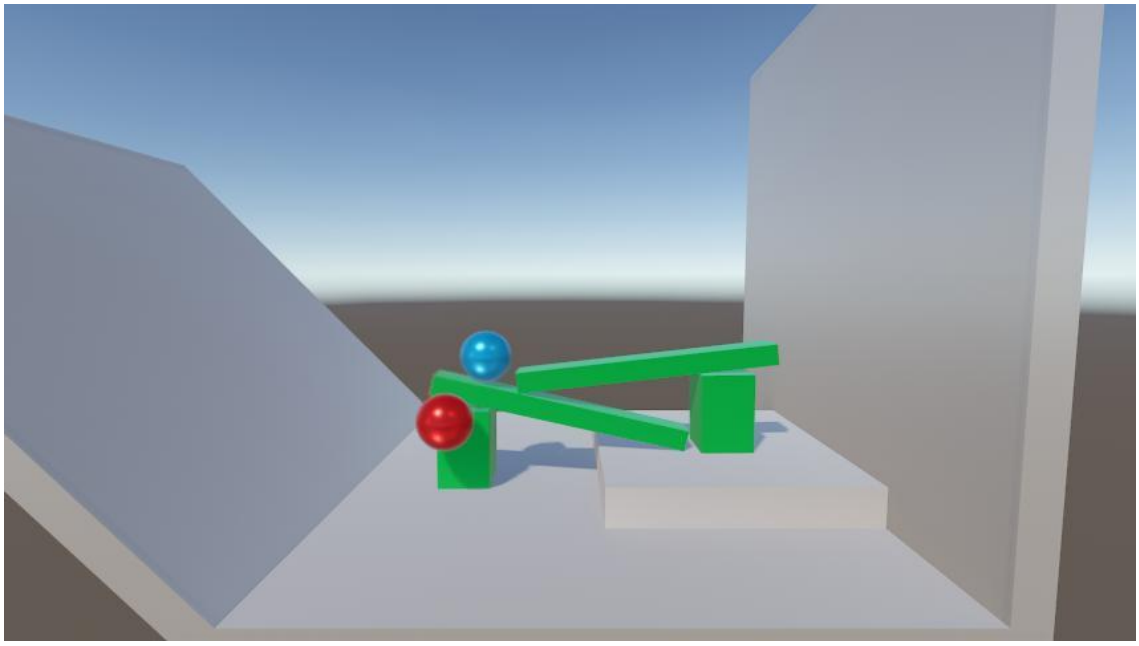
Kuva 11. Esittelysovelluksen alkuasetelma. Vihreät laatikot sekä punainen ja sininen pallo ovat dynaamisia jäykkiä kappaleita. Harmaat laatikot ovat staattisia kappaleita.

Kuvassa 12 nähdään, kuinka dynaamiset kappaleet liikkuvat alaspäin painovoiman suuntaan staattisten kappaleiden pysyessä täysin liikkumattomina. Kappaleiden törmätessä laatikkomaiset kappaleet kiertyvät ja asettuvat päällekkäin pinoon muodostaen vinon liikkumapinnan pallomaisille kappaleille ja saaden ne vierimään toisiaan kohti.



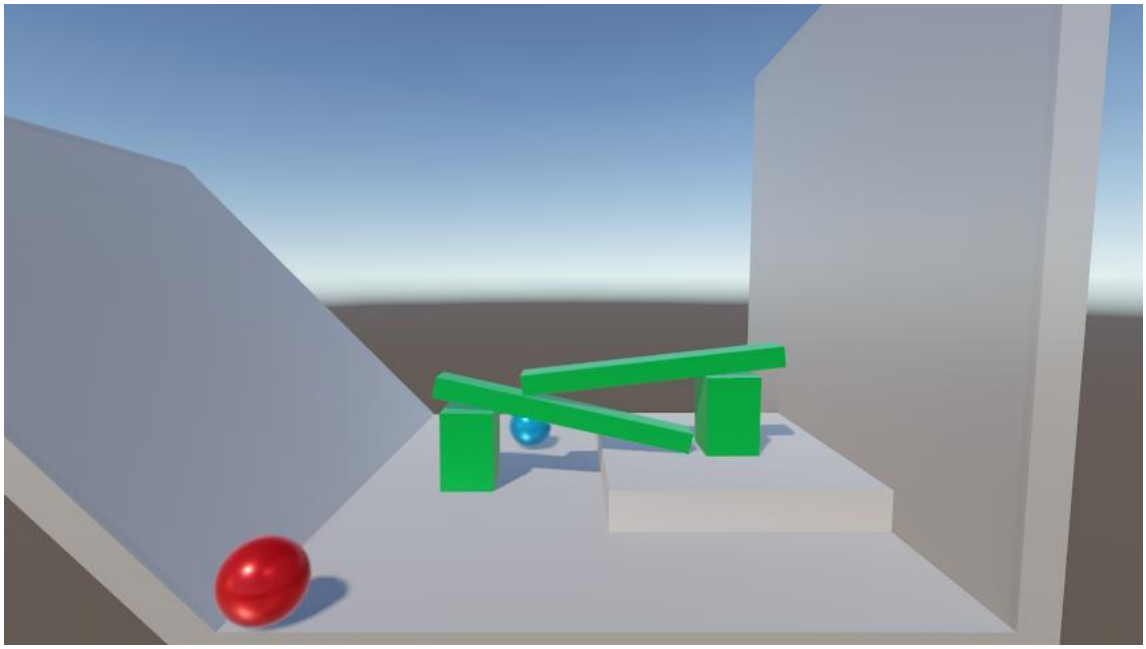
Kuva 12. Usean kappaleen välinen törmäys esittelysovelluksessa.

Kuvassa 13 nähdään, kuinka pallojen törmätessä massaltaan suurempi sininen pallo tönäisee kevyempää punaista palloa ja saa sen vierimään alas pinosta.



Kuva 13. Pallojen välinen törmäys esittelysovelluksessa. Painavampi pallo tönäisee kevyemmän alas laatikkopinon päältä.

Kuvassa 14 nähdään, kuinka esittelysovelluksen lopussa laatikkomaiset kappaleet pysyvät yhä kitkan vaikutuksen takia suhteellisen liikkumattomina pallojen pudotessa ensin alustalle ja lopulta vieriessä kohti sen reunoja.



Kuva 14. Esittelysovellus hetkeä myöhemmin kappaleiden törmättyä.

Esittelysovellus osoittaa, että fysiikkamoottorilla nykyisessä muodossaan on mahdollista simuloida usean jäykän kappaleen keskinäisiä vuorovaikutuksia jokseenkin uskottavalla tavalla, eli se täyttää tältä osin sille työssä määritetyt vaatimukset. Nähtävissä on myös, että moottori suoriutuu kohtalaisesti myös pinoamisesta pienellä määrällä kappaleita. Tämä on useimmiten riittävän monimutkainen simulaatioskenaario yksinkertaisiin pelisovelluksiin, mutta hyvin fysiikkapainotteisissa peleissä saatetaan tarvita tukea huomattavasti suuremmille määrille kappaleita.

Testatessa paljon vaativampia skenaarioita alkoi fysiikkamoottorin vakaus ja suorituskyky kärsiä nopeasti. Kun testattiin yli kuuden päällekkäisen kappaleen pinoja, ilmeni selkeästi näkyvää uppoamista kappaleiden välillä sekä useimmiten ennen pitkää kappaleisiin kohdistuvia epärealistisia sinkoavia voimia. Näihin ongelmiin ei vielä työn kirjoitushetkellä löytynyt ratkaisua, joten ne jäävät selvitettäviksi jatkokehitystä varten.

# 8 Yhteenveto

Insinöörityössä saatujen tulosten pohjalta voidaan todeta, että työn tavoitteet saavutettiin kohtalaisesti. Testeissä havaittiin, että kappaleet simulaatiossa liikkuvat ja törmäävät useimmissa tapauksissa fysikaalisesti uskottavasti. Joissakin testeissä kappaleet kuitenkin käyttäytyivät odottamattomalla tavalla, eikä moottorin suoritusteho ole useissa tapauksissa optimaalinen. Nykyisessä muodossaan fysiikkamoottori on kuitenkin tarpeeksi vakaa soveltuakseen käytettäväksi yksinkertaisissa pelisovelluksissa. Moottori on ominaisuuksiensa puolesta pelkkä runko, mutta työn yhteydessä luodut kirjastot ja järjestelmät mahdollistavat moottorin laajentamisen tulevaisuudessa.

Fysiikkamoottoria varten tehty tutkimustyö opetti todella paljon pelien fysiikkamoottorien toiminnasta ja niiden kehitykseen liittyvistä haasteista, ja tätä kautta se myös huomattavasti avarsi mahdollisuuksia moottorin jatkokehitystä varten. Mahdollisia lisättäviä ominaisuuksia ovat muun muassa tuki kappaleita yhdistäville nivelille, kangas- ja nestesimulaatiot sekä pehmeät kappaleet. Koska jäykkien kappaleiden välisten törmäysten simulointi kuitenkin on moottorin keskeisin käyttötarkoitus, on nykyisten järjestelmien hiominen oleellisinta jatkokehityksen kannalta. Erityisesti törmäysten tunnistuksen optimoiminen ja törmäysten ratkaisemisen kehittäminen vakaammaksi tekisivät fysiikkamoottorista varteenotettavamman vaihtoehdon käytettäväksi todellisissa peliprojekteissa.

## Lähteet


1 Bywalec, Bryan & Bourg, David M. 2013. Physics for Game Developers. 2. painos. E-kirja. O'Reilly Media.

2 Eberly, David H. 2010. Game Physics. E-kirja. Burlington Morgan Kaufmann.

3 Souto, Nilson. Video Game Physics Tutorial - Part I: An Introduction to Rigid Body Dynamics. Verkkoaineisto. <https://www.toptal.com/game/video-game-physics-part-i-an-introduction-to-rigid-body-dynamics>. Luettu 22.2.2020.

4 Serrano, Harold. 2016. How does a Physics Engine work? An Overview. Verkkoaineisto. <https://www.haroldserrano.com/blog/how-a-physics-engine-works-an-overview>. 24.4.2016. Luettu 15.2.2020.

5 Szauer, Gabor. 2017. Game Physics Cookbook. E-kirja. Packt Publishing.

6 Parent, Rick. 2012. Computer Animation. 3. painos. E-kirja. Morgan Kaufmann.

7 Pharr, Matt; Humphreys, Greg & Jakob, Wenzel. 2004. Physically Based Rendering. 3. painos. E-kirja. Morgan Kaufmann.

8 Menache, Alberto. 2011. Understanding Motion Capture for Computer Animation. 2. painos. E-kirja. Morgan Kaufmann.

9 McGinty, Bob. Rotation Matrices. Verkkoaineisto. <https://www.continuummechanics.org/rotationmatrix.html>. Luettu 6.5.2020.

10 Matrix4x4.Rotate. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/ScriptReference/Matrix4x4.Rotate.html>. Luettu 6.5.2020.

11 Fiedler, Glenn. 2004. Fix Your Timestep! Verkkoaineisto. <https://gafferongames.com/post/fix_your_timestep/>. 10.6.2004. Luettu 12.2.2020.

12 Gaul, Randy. 2013. How to Create a Custom 2D Physics Engine: The Core Engine. Verkkoaineisto. <https://gamedevelopment.tutsplus.com/tutorials/how-to-create-a-custom-2d-physics-engine-the-core-engine--gamedev-7493>. 1.5.2013. Luettu 12.2.2020.

13 Vector3.Lerp. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/ScriptReference/Vector3.Lerp.html>. Luettu 26.4.2020.

14 Quaternion.Slerp. Verkkoaineisto. Unity Technologies. <https://docs.unity3d.com/ScriptReference/Quaternion.Slerp.html>. Luettu 26.4.2020.

15 Fiedler, Glenn. 2004. Integration Basics. Verkkoaineisto. <https://gaf-ferongames.com/post/integration_basics/>. 1.6.2004. Luettu 13.2.2020.

16 Rodriguez, Jorge. 2016. Numerical Methods for Physics Integration in Video Games. Verkkoaineisto. <http://www.vinoisnotouzo.com/blog/10.htm>. 9.7.2016. Luettu 13.2.2020.

17 Ericson, Christer. 2004. Real-Time Collision Detection. E-kirja. Burlington Morgan Kaufmann.

18 Souto, Nilson. Video Game Physics Tutorial - Part II: Collision Detection for Solid Objects. Verkkoaineisto. <https://www.toptal.com/game/video-game-physics-part-ii-collision-detection-for-solid-objects>. Luettu 23.2.2020.

19 Charry, Jamie. 2017. Building a Physics Engine, Pt. 10 - Collision Detection using Separating Axis Theorem (SAT). Verkkoaineisto. <https://jcharry.com/blog/physengine10>. 27.1.2017. Luettu 12.3.2020.

20 SAT (Separating Axis Theorem). 2010. Verkkoaineisto. dyn4j.org. <http://www.dyn4j.org/2010/01/sat/>. 1.1.2010. Luettu 12.3.2020.

21 GJK (Gilbert–Johnson–Keerthi). 2010. Verkkoaineisto. dyn4j.org. <http://www.dyn4j.org/2010/04/gjk-gilbert-johnson-keerthi/>. 13.4.2010. Luettu 14.5.2020.

22 Gaul, Randy. 2013. How to Create a Custom 2D Physics Engine: The Basics and Impulse Resolution. Verkkoaineisto. <https://gamedevelopment.tutsplus.com/tutorials/how-to-create-a-custom-2d-physics-engine-the-basics-and-impulse-resolution--gamedev-6331>. 6.4.2013. Luettu 20.3.2020.

23 Gaul, Randy. 2013. How to Create a Custom 2D Physics Engine: Friction, Scene and Jump Table. Verkkoaineisto. <https://gamedevelopment.tutsplus.com/tutorials/how-to-create-a-custom-2d-physics-engine-friction-scene-and-jump-table--gamedev-7756>. 15.5.2013. Luettu 20.3.2020.